UNIVERSITY OF CALIFORNIA
SANTA CRUZ

A STUDY OF FRAMEWORKS FOR COLLECTIVELY ACHIEVING THE PRODUCTIVITY, PORTABILITY, AND ADOPTABILITY GOALS OF PARALLEL SOFTWARE

A dissertation submitted in partial satisfaction of the
requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER ENGINEERING

by

Sean Halle

June 2011

The Dissertation of Sean Halle
is approved:

Professor Jose Renau, Chair

Professor Cormac Flanagan

Professor Albert Cohen

Tyrus Miller
Vice Provost and Dean of Graduate Studies

Contents

List of Figures

List of Tables

Abstract

A Study of Frameworks for Collectively Achieving the Productivity, Portability, and Adoptability Goals of Parallel Software

by

Sean Halle

The search for a parallel programming language that is both highly productive and high performance across a wide range of hardware has been in progress since the 1960's, when multi-programming became popular. That's nearly 50 years, without satisfactory success. The recent move to multi-core processors has accelerated the need for a broadly adopted solution.

This dissertation suggests that the shortcoming stems from the complexity of the triple goal of productivity, portability, and adoptability. These involve many real-world phenomena that have no scholarly studies to rely upon. In this situation, the standard research practice of simplifying a problem to the point that a defendable result can be shown is counter-productive, because work progresses down paths that have no hope of success and can't be reused when more successful paths are found. This drains resources away from more promising paths. Too many interdependencies, across too vast a cross-section of fields, are in play, so that research that solves a simplified problem doesn't translate well to the real world. More importantly it doesn't add constructively to other, independent, research.

This dissertation targets the triple goal, of productivity, portability, and adoptability, by taking the position that infrastructure is needed that takes into account the full complexity of the real-world problem, and acts as a guide to make independent research efforts coherent and cumulative. Over time, the infrastructure will transition from a research helper to a standard part of the software landscape.

Such infrastructure will be termed a framework in this dissertation. In effect, the framework constrains the simplifying assumptions, with the result that research aligned with those constraints works together, additively, with other aligned research. Hence, with the framework in place, the traditional approach can be followed by individual groups, and collectively an overall solution to the triple challenge will emerge, with pieces contributed by far-flung, unconnected groups. The framework acts as an organizing principle, providing coherence of independent research efforts.

The first part of this dissertation discusses the nature of the problem. It is not scholarly, due to the complexity and imprecision of the real-world aspects. It is best viewed as a first step in an iterative process: a try is made at informally modeling the problem, followed by proof-of-concept solutions aligned with that model, that motives the next iteration of more objective and thorough observations to create a better model, followed by better solutions guided by the new model, and so on. Such iteration is the standard approach to breaking cyclic dependencies, and this dissertation takes the position that it is a good time to try adding such informal models of real-world phenomena that affect productivity and adoption.

The adoptability goal is particularly tied to real-world details, especially of how software is developed and distributed. In practice, this happens in many different ways, and the major practices need to be understood in order to design a framework compatible with them.

Therefore, the discussion of the problem begins by categorizing software into segments, whose boundaries are identified, in part, by development practices. The characteristics of each segment are discussed and a subset of them chosen as the targets to inform the framework design.

The portability goal is modelled with a computation model, designed to expose the structure of parallel computation related to portability. It is used as the base upon which to analyze parallelism's structure, from the application view, from the hardware view, and the

mapping between them.

This dissertation defines the mapping of application onto hardware as porting, and the subset of that related to performance as the specialization process. The analysis of parallel computation exposes what the specialization process needs as input to produce a highly efficient mapping.

The parallel-computation model's main outcomes are the activities to be performed during specialization, and the consequent needs of specialization. The activities include identifying boundaries of parallel tasks in the application, modifying the structure of the application to make the resultant tasks and their communication patterns fit efficiently onto the hardware, inserting a runtime implementation tuned to the target hardware, and low-level optimizations of the task-code. These activities need the language to expose tasks, the dependencies between them, certain properties of the tasks, and to prevent scheduling code from appearing in the application.

The second part of this dissertation goes on to relate two candidate frameworks that were designed in accordance with the model and analysis. Each candidate framework has a working proof-of-concept prototype. The first, called BLISS, is based on a centralized server that performs specialization, after development but before distributing the installable image. The second is based on the *morphable* hardware abstraction called VMS. VMS has no application-usable semantics, but rather accepts those in the form of a plugin, which makes it morphable. It is a direct replacement for the Thread hardware abstraction. And, it is suitable as the basis for a portability framework because it breaks the specialization process into three independent steps, allowing separate entities to perform those steps independently. Hence, no centralized server is needed, and the decomposition increases reuse of effort put into specialization tools.

These frameworks provide a road for reusable tools to deliver performance portability of the widest possible range of applications, written in highly productive languages and envi-

ronments. The frameworks fit various aspects of the software segments, addressing adoptability issues that have blocked uptake of other approaches. Rather than try to create the languages and specialization tools itself, the frameworks instead provide standard interfaces and methodology to support other, unconnected groups, to develop the languages and tools.

This dissertation does not claim a solution to productivity nor to portability directly. Rather it claims something to *guide the search* for a solution, organizing the many independent research groups, to provide accumulation and coherence of their efforts. It will eventually transition to standard infrastructure in suitable software segments.

This dissertation's results should be measured by whether they make the search for a solution faster, whether they make search results coherent with each other, and whether they appear compatible with industry, having a reasonable path to adoption as standard infrastructure.

Some initial tests have been performed with the frameworks to gain an indication of how well they might support a collective search. For the embedded segment, performance is top priority, and a poor performing framework would block adoptability, so performance numbers are given for proof-of-concept languages built on top of the frameworks. These show for each framework that it does not present a performance barrier to adoption. A modest productivity study for creating new languages based on VMS is also presented, and shows that the search for new parallel languages, such as embedded domain-specific languages, is accelerated by VMS. A proof-of-concept specialization server is demonstrated in BLISS, which shows that automated specialization can successfully take place in a centralized server. This in turn shows that BLISS provides the necessary conditions for specialization, and hence for portability.

To Tiny's

Where I felt the Love while developing the basic theories

and to Judy,

for trading chicken-coop for rent and taking good care of me while writing :)

and to Gabe,

who may be the only person to actually read this – even if he doesn't answer my emails ;)

Acknowledgments

I want to thank my committee. It's a lot of work, without much reward, so I am grateful to Jose, Cormac, and Albert for being here for me. Thank you.

I would like to thank Jose for his flexibility and hanging in there with me to the end, it would have been very easy for him to throw up his hands and walk away. I'm grateful that he has stuck it out.

In addition, I want to give a special thank you to Albert, who took a chance on me, and I hope that someday I will be able to repay my debt to him. The experience of working together has been extraordinary. He has been a friend, a mentor, and an engaging co-conspirator. And always a pure joy to spend time with. Our few disagreements may have been the best part, challenging both of us to interesting new ways of thinking.

This has been a difficult span of time, in no small part due to UCSC, which has some deeply needed improvements to make in its administrative and graduate approach. Here's hoping the balance shifts just a tad, so no one else has to go through having it quite this bad.

Getting me through these years have been my awesome friends. The highlights were time spent with Dan, Yulia, Gabe, Alex, Judy, Bill, Julie, Lief, Dmitry, Ozana, Arnaldo, Holger, Kaira, my Tiny's Family, and most especially Kalen, and my son Leander. I love you all (but you can skip the hugs, you know who you are ;)

Chapter 1

Motivation and Overview

> One cannot understand the solution until one understands the problem. – proverb
>
> Make everything as simple as possible. But not simpler. – Albert Einstein

Parallelism was first exploited as a means to allow many users to share a single machine. It was then adopted for scientific problems because it was the means to obtain the highest possible performance[50]. But the experience of writing parallel code has proven much more difficult than writing sequential code, despite nearly 50 years of effort to make parallel programming the same level of difficulty as sequential programming. In addition, parallel code has proven more expensive than sequential code because it has been necessary to modify the source when it is run on a different machine, in order to get acceptable performance on the new machine.

The problems: These two problems: the difficulty of writing parallel source code, and the necessity to change it for each machine, have prevented parallel programming from being accepted into the mainstream. This in turn has contributed to stopping the advantages of parallel hardware from being widely available. Parallel processor chips have been researched and even appeared commercially for niche markets, but the difficulty of programming them and the lack of good-performance portability of the code to other processors kept them out of the

mainstream.

The driving force: Within the last half decade, this has been changing because the energy efficiency advantages of parallel hardware has forced its adoption for mainstream processors. As a result, the need for high productivity languages for writing parallel code and the need for that code to be portable, with good performance, to future processor chips has reached a critical level. Indeed, the economic cost will be immense if these productivity and portability challenges fail to be satisfied, in a way that is readily adoptable by industry.

Past attempts: In an effort to solve the productivity and portability challenges, hundreds of parallel languages and libraries have been invented and thousands of papers written on research relating to the topic. Despite this effort, so far, widely acceptable solutions continue to elude researchers and industry.

Block to past attempts: This dissertation suggests that the shortcoming stems from the complexity of the triple goal of productivity, portability, and adoptability. These involve many real-world phenomena that have few scholarly studies, and no comprehensive ones, to rely upon. Involved are psychology of decision making, business practices, programmer culture, and many other non-precise areas, for which scholarly studies are difficult. Numerous books have been written about these areas, but of an informal nature.

In this situation, the standard research practice of simplifying a problem, to the point that a defendable result can be shown, is counter-productive, because it sends researchers down paths that have no hope of success, and their work ends up in a form that doesn't apply to the promising paths.

The simplifying assumptions create a model of the problem that is incomplete. Solving the triple challenge involves balancing competing goals against each other. A model that

simplifies by eliminating, for example, adoptability and productivity concerns, will lead the researcher to solve the portability problem, but in a way that the majority of programmers find bewildering, and hence is unlikely to be adopted outside of research.

This dissertation's way around the block: Such simplifications in research are time honored and work in most situations, but after so many years of trying that approach for parallelism, this dissertation argues that now would be a good time to try accepting informal models of the problem into academic research. The full first part of this dissertation does just that, giving a model of industry segments, with development and decision making practices in each that are relevant to adoption and productivity. The model is meant as a guide to use while designing potential solutions to the triple challenge.

Reason to make a framework: The first major outcome of the model is the suggestion that the problem is beyond the reach of a single research entity. Portability requires multiple real-world entities to work together, namely hardware-focused, language-focused, and application-focused research groups. In addition, deep technical expertise is required in compiler analysis and transform, in schedulers for particular hardware architectures, in language design, and in tools for development, build and distribution. It is unlikely that a single research group will have all the requisite expertise, much less the budget to complete the work.

The second major outcome is the need for a tool that allows separate real-world entities to coherently contribute. The dual needs of being able to work independently and also integrate with each other can only be satisfied by the integration being encoded in a framework. The independent work fits into slots within the framework, then the framework makes that work integrate with each other.

Concrete contribution of dissertation: In accordance with this, two frameworks have been designed and proofs-of-concept for them implemented. They both take into account the full complexity of the real-world problem, as modelled in part 1 of this dissertation.

How contribution sidesteps block: Rather than try to solve the problem directly, each framework instead acts as a guide to make independent research coherent and cumulative. Each achieves this coherence in a different way: the first by organizing around a centralized server that defines standard interfaces and patterns; the second by organizing around a mutable hardware abstraction that cleanly separates application from scheduling and those from hardware.

With such a framework in place, the traditional approach of simplifying the problem can be followed by individual groups, but the simplifications are supplied by the framework. The framework provides a context that makes these independent efforts integrate, so collectively an overall solution will emerge. The framework acts as an organizing principle, providing coherence of independent research efforts.

Preview of Dissertation Chapters: The first part of the dissertation has seven chapters. The first, Chapter 2, discusses structure within software, in a top-down way, grouping software into segments according to characteristics it experiences over its lifetime. The second, Chapter 3, is bottom-up, taking a personal view, discussing how people handle parallelism in the real world and what clues that might give for future language design. It then discusses individual preferences among developers, closing with the implications from the variations among people and how that will affect productivity and adoptability. Chapter 4 then takes the point of view of the decision process for which language and toolset get used on a given software project. This illustrates how new languages, development processes, and the infrastructure that supports them get adopted.

The next four chapters cover portability, starting with a chapter that overviews the three following it. The first of those, Chapter 6, introduces a basic model of computation, the next uses it to analyze parallelism, and the third, Chapter 8 uses the first two as context to analyze what is involved in making given source run high performance on particular target hardware, and what an automated process for that needs. This has implications for language design as well as design of a framework that supports specialization effectively for a broad array of languages, development processes, application types, and hardware types.

The second part has 5 chapters. The first, Chapter 9 covers the first framework, BLISS, the second, Chapter 10, covers the mutable hardware abstraction, VMS, the third shows how to build a framework on top of VMS, the fourth, Chapter 12, compares and contrasts the proposed frameworks with other approaches to the triple challenge. And the fifth and final, Chapter 13, concludes the dissertation with a brief summary of the main takeaway points.

Part I

Model of the Problem

In order to understand what is needed to solve the productivity and adoption challenges, the relevant aspects of the world in which software is produced, distributed, and run should be modelled. This Part 1 does just that. It also gives a mental picture of what parallelism is, and what is involved in making given source run efficiently on given hardware, with given input data.

The real-world aspects related to productivity and adoption are imprecise, so this first part will not be scholarly nor backed by references to published results, but rather a summary derived from years spent working in several segments of the software industry and being exposed to others. As such, it contains subjectivity.

1.0.1 Dealing with Imprecision and Subjectivity

The subjectivity presents a dilemma. After admitting that suitable objective measurements of these areas haven't been found, one choice is to insist on *only* taking into account objective measurements. The consequence is causing the imprecise areas to be left out of the model of the problem. This equates to a significant simplification.

Because productivity and adoptability are two of the triple goals, simplifying away real world aspects that relate to them is counter productive, tending to encourage work that provides poor or no value. Productivity and adoption are a function of people: how they work, how they think, and how they make decisions. Modelling these is imprecise and subjective, but having no information about them in the model is even more imprecise. It maximizes the uncertainty in the model on these aspects, making the model less useful, perhaps deceptively so.

Therefore, the approach taken in this dissertation is to talk to many people involved in these real-world areas and gather as much observation as possible. All of the information gathering was done informally. Although this injects uncertainty into the research, it is expected to be a better model of the problem than one with zero information. The real world itself will

test whether this approach is a good one, by adopting some descendent of the research, or not.

The informal model is a first iteration of an ongoing process. Hopefully the work based on this informal model will show success and inspire more objective studies of these practices to base further iterations of frameworks upon.

Chapter 2

Software Segments

This chapter takes the point of view of a software application, and talks about what it experiences over its lifetime. This is used to identify segments of software, by grouping software with similar life-experiences. The value of this lies in grouping code-projects together that have common influences on productivity and adoptability.

The influences include the infrastructure surrounding the code at each phase of its lifetime, the language the code is in terms of, the development process and stages of development, and the decision making priorities that drive those choices.

2.1 Infrastructure Surrounding Applications

Applications have a life-cycle during which they are surrounded by infrastructure, which includes a development environment while it is being created, a distribution system to get it from developer to the machine(s) it runs on, and a software-stack when it is running. Software can be divided into segments based on the properties of the infrastructure it experiences over its lifetime.

How relevant to adoption: Such infrastructure is relevant to adoption, because the proposed frameworks will become part of this infrastructure, if adopted. Existing infrastructure should inform the design of new infrastructure to be added-on, in order to minimize barriers to adoption of the new infrastructure. As will be seen in Chapter 4, which models adoption, coherence with existing software development workflows and practices reduces barrier to adoption.

Multiple frameworks: What works in one segment may not work in another, so one framework may be appropriate in some segments, while a different framework is best in others. First, though, the characteristics and structure of the segments must become known and understood.

Development environment: The first bit of infrastructure experienced is the development environment used to initially create the application. Later revision or updates also experience a development environment. The second bit experienced is the distribution system by which the application makes it onto the hardware it runs on. The last bit is the software-stack, which provides the context in which an application runs.

A Development Environment consists of the language(s) used to write the application, APIs available from the software stack, the development tools used while a programmer codes, the quality assurance tools and process, the workflow used by the team, and the development process used by the organization. Many of the tools used will enforce or enable parts of a development process. Some segments commonly have no formal development process, others tend to have complex and sophisticated ones, others are a mix. In general, the development environment varies on two axes: by segment and within that by development entity.

Distribution: The Distribution system is the means for the installable image to get from developer to machine it is installed and runs on. This varies by segment. It is most important for shrink-wrap, where sales are affected by the method of delivery to the consumer, and convenience

for the purchaser is highest priority. At the other extreme, for web-sites, distribution doesn't exist as a separate function because the software runs on the same machine it is developed on (or is under the control of the developer). The distribution process has implications for adoptability of proposed frameworks, and is especially relevant to the suitability of the BLISS approach.

Software-stack: The software-stack is what an application interacts with while it is running, and includes the Operating System, static libraries that get inserted into the executable, dynamic libraries, and interacting applications and utilities. Each segment has common characteristics of the software stack.

Although an application may see one software stack during development, Chapter 8, covering specialization, will show that by the time the application reaches the target hardware and runs, the software stack may have completely changed.

Group by infrastructure: Next, software will be grouped into segments, according to commonalities in infrastructure seen over their lifetimes. The segments tend to be aligned with sectors of industry, because forces within the industry drive choices for the infrastructure. But this is not always the case. For example, Open Source is considered a separate segment because the development tools, distribution process, and software stack are relatively common for the majority of Open Source software, despite Open Source spanning multiple industry sectors.

2.2 The Software Segments

This dissertation divides software into segments by grouping by the characteristics of the infrastructure that surrounds an application over the stages of its life-cycle. These segments are:

- embedded software

- web software
- enterprise software
- shrink-wrap software
- mobile applications
- system software and toolchains
- Open Source software
- Scientific Computing

Segments chosen to drive framework design: For this dissertation, the priority was placed on the embedded, enterprise, shrink-wrap, and mobile segments. They informed the design of the frameworks described in part 2. This is because the web segment has no great need for a parallel language in the near future; system software and toolchains are highly specialized, have few programmers, and are likely to employ their own custom solutions; scientific has a tiny fraction of programmers; and Open Source has too little organization or coherent need to serve as a useful target. However, Open Source may be the first segment to adopt the frameworks or descendants, so it is not discounted in importance, but rather wasn't considered a relevant source of guiding observations.

Focus on largest number of programmers: From a research perspective, the chosen segments may not appear to be the most important. This is because, a disproportionate amount of funding is supplied to public research institutions for particular sub-segments, such as scientific computing, and perhaps safety-critical embedded. A survey of job listings indicates that these segments have a tiny fraction of a percent of total programmers. Therefore, in keeping with the

adoption goal, this dissertation chooses to focus instead on the segments with the largest total of programmers.

2.2.1 Listing of the Segments

This subsection describes each segment listed above. For each segment, first it defines what software falls within it and summarizes its priorities. Then, it details the development environment, distribution process, software stack, and driving factors in development.

2.2.1.1 Embedded Segment:

This segment includes software put onto micro-controllers and boards inserted into consumer products, plus BIOS development, and any place where the software is not distributed separately, but rather ships as part of a product. Absolute performance is the driving priority, followed by time-to-market, and cost to introduce or adopt a new chip.

Development Environment The dominant language here is C followed by various assembly. A large percentage of developers use text-based tools rather than visual ones. Many companies have custom development tools and workflows. Verification tools are central to development for safety-critical embedded products like airplanes and automobiles, as well as for many large-volume consumer products like televisions.

Distribution The applications have no separate distribution process because the software ships inside the product.

Software Stack is flexible and often custom to a particular chip or OEM vendor

Driving factors in development Absolute performance has higher weight than productivity or functionality. This is because cost of CPU and power consumption are normally economic drivers, so the most has to be gotten out of limited computation resources, and C

has proven good at that. Reliability is also a major factor due to high cost of bugs in the shipped product. Code bases tend to be small, strong quality assurance and verification processes are typically used.

2.2.1.2 Web Segment:

Web software includes the front-end portion of a corporate web-presence, sites that are a company's entire business in and of themselves, such as Facebook, and content portals, such as iTunes for music, Salon.com for journalism or Hulu for video, and web-stores. Search engines are special-case, meta, web-sites. The overwhelming priorities are productivity and interchangeability of programmers, who often work on a code base on a temporary basis, and/or jump around to different parts. A common phrase in this segment is "hardware is cheap."

Development Environment The most popular languages here are PHP, Java, Javascript, and visual tools. Java is popular for sites that generate HTML on the fly, especially ones with large amounts of business logic or other back-end processing on the server. If a large amount of business logic exists, that part of the site is often considered to fall within the enterprise segment, depending on size and purpose.

Distribution There is no distribution process because the software runs on the same server it is developed on, or is copied by the development team to a similar server.

Software Stack A web-server software stack, either commercial or open-source

Driving Factors Productivity is nearly the only consideration here because such web-sites don't tend to perform large amounts of intensive computation, more hardware can be easily added, and the software is much more expensive to develop than the hardware is to buy, by a many to 1 factor. The code base tends to be modest (under 100K lines) and is

often completely replaced over a span of years. The programmer-team size for front-ends is typically small, and development process varies quite widely.

2.2.1.3 Enterprise Software Segment:

This software overlaps with the Web segment because it nearly always has a web front-end. However, in this segment, access is often limited to a company's internal network. This segment includes traditional enterprise functions such as accounts, billing, payroll, and customer management. It also includes any software that a company develops for its own use either internally as part of how the company is run, or externally as a tool used by customers in some aspect of managing their account and purchasing. The overriding priority is development process, allowing programmers to come and go and functionality to be updated in a methodical and predictable way over the many years of a code-base's lifetime. Predictability of development tends to trump features and efficiency. The same code-base may transition from Web to Enterprise over time, as it grows and the company matures and process becomes more important.

Development Environment The dominant language is Java, almost exclusively. A variety of workflow and development process tools are used, often custom to the development organization. Strong development processes are enforced and supported by a variety of development tools, some custom created by the development entity.

Distribution Same as in Web development.

Software Stack Often includes middleware and service-added packages for CRM, billing, and so on, that provide base functionality that is then customized.

Driving Factors The same entity that develops the application also controls the machines it runs on and is the only entity that uses it. The developers either work for this same

company or for a consulting company that develops the software to spec. Performance has low weight because computation resources are easy to add. More importantly, the development cost of the code is not amortized over many units sold, so productivity is paramount. The most important feature is that the projects in this segment are the largest of all, reaching into hundreds of millions of lines of code, with hundreds to thousands of programmers. Sophisticated software development processes are used here, to manage the complexity of the code base and knowledge transfer between programmers over the decades of a code base's lifetime.

2.2.1.4 Shrink-wrap Segment:

This includes any off-the-shelf software package that can be purchased by a third party, installed on their machine, and used. This includes games, CAD tools, office-suites, and so on. The priority is on balancing: features, low bug count, performance, and high productivity. It has the most competing priorities, having to be balanced, of any of the segments. Development process tends to vary with size of the code-base and maturity of the entity creating the code.

Development Environment The dominant languages are C++, and Java.

Distribution Part of the retail process. Either download via web, or physical media such as DVD. Must be convenient and simple for the customer.

Software Stack Windows and MacOS are the most popular. Libraries either come with the OS or with commercial development tool suites. (Note that Open Source software is not included here, and commercial shrink-wrap for Linux is insignificant in volume)

Driving Factors Here, productivity is balanced against performance. Both matter because products are compared against each other for performance, but also for functionality and price, which are consequences of productivity. The development practices are often custom

to a particular software-vendor, and span from ad-hoc to sophisticated. Team sizes also vary. Distribution is part of the retail process and so is important.

2.2.1.5 Service-Driven Software Hybrid Segment:

There is a hybrid segment bridging between shrink-wrap and Enterprise. This is where service-driven-software lies, which has core functionality but is then customized to a customer's needs. PeopleSoft is an example, as is Siebel customer management, Vitria middle-ware, and Database systems. The creation of the base package falls in the low-volume shrink-wrap segment, but the customization for the customer falls in enterprise development. For these packages, the package itself is a form of domain-specific language, and customization often involves the difficult issues associated with distributed systems. The teams that customize these span from individuals up to dozens of programmers, and tend to be employed by consultant shops. After customization, level of activity drops but maintenance to upgrade and add features still takes place. These shops often have well-enforced development processes due to the need for predictable development times, high programmer turn-over and the need for repeated, reliable, adaptable development.

2.2.1.6 Mobile Application Segment:

This is newly emerging and may grow into a dominant segment. Applications are for mobile devices like smart-phones and Personal Digital Assistants. The code-bases are very small and typically developed by one to only a few people. So far, development processes appear to be ad-hoc for the majority of these applications. The driving factor is features, which have to conveniently address a specific need. Behind this is productivity, especially for individual programmers or small teams. The code-base tends to be re-written rather than evolve and grow over many years the way enterprise and to a lesser extent shrink-wrap do.

Development Environment These tend to be small projects with relatively few lines of code,

and no currently dominant language nor development process or style.

Distribution The software is downloaded from a server as part of an automated purchase process. The connectivity service-provider often owns and maintains the server, collecting executables from 3rd party developers.

Software Stack mobile devices have a wide variety of operating systems and supporting software stacks. This creates an issue for developers who have to either pick a platform or else spend considerably more to develop cross-platform. The business issues are complex, between service providers, device supplier, and developers.

Driving Factors mobile devices are a key area for parallelism due to the higher efficiency of parallel hardware, and so of special importance in the subject of this dissertation. Parallel hardware has been slow in adoption due to the programmability barrier.

2.2.1.7 System Software and Toolchain Segment:

This is where the infrastructure is created. It is rather small in terms of the total number of programmers, and is a meta-segment, responsible for creating the infrastructure for the other segments (as well as for itself). Development of the OS, libraries, and utilities that make up the software stack fall in this segment. The developers here are the ones who create the highly productive and portable languages, rather than being targets for using them.

Development Environment The dominant languages are C, C++, and Java.

Distribution varies, no consistent process

Software Stack This segment creates much of the software stack itself

Driving Factors The needs of developers in these areas are specialized, and the types of developers tend to be very highly skilled and creative. Reliability of final code, bug-free is

strongly desired. Verification is important.

2.2.1.8 Open Source Segment:

This is more of an umbrella segment, or shadow segment, with equivalent applications as in all the other segments. It differs in that it has much less or no economic drive to create the applications, with a few notable exceptions. The applications attempt to solve the same problems as the commercial counterparts, but have very different development processes and software stacks. The driving force is learning, fun, and sense of control/empowerment. Development processes tend to be ad-hoc, distributed, and weak on quality assurance and documentation.

Development Environment Open Source is normally developed in very different ways than the commercial versions, which sets it apart when trying to understand common practices in software development. Open Source uses the widest range of languages, has a wide array of development processes and styles, and uses a wide range of development tools.

Distribution is almost uniformly over the web, via automated process invoked from a web page. Sometimes only source is available, but for the most popular OSes, installation bundles are often available that include all the elements of the software stack needed for the application. A few servers hold the vast majority of the open sources software (Sourceforge, GitHub, and so on).

Software Stack the OS is almost exclusively Linux, but libraries have been developed that create the needed Linux interface on Windows and other commercial OSes. The supporting software stack varies widely between applications. However, automated systems for obtaining the needed support libraries for a given application have been developed. Some embedded software is released Open Source, but is not yet common. This segment has little coherence, other than the distribution process.

Driving Factors Software in this category is normally done for learning, with a sub-category released Open Source to drive a dual-license or services business. MySQL is a dual-license example, and RedHat is a services example. In general, no commercial concerns drive development, but rather personal interest in some aspect of software. Productivity is a strong driver, due to the volunteer nature, and teams tend to be small and loosely organized, mainly by email lists. Notable exceptions exist, such as Eclipse which is supported by IBM and widely used to create custom development tools for commercial concerns, especially in the embedded segment. The frameworks described in part 2 fit many patterns of Eclipse, which shows a promising path for adoption.

2.2.1.9 Scientific Segment:

This is the oldest parallel software segment. It tends to be dominated by relatively easy-to-parallelize applications whose performance is concentrated in a modest number of small kernels. Driving factors are verification of correctness and performance, followed distantly by productivity. Many of the kernels from this segment migrate to embedded and shrink-wrap, and may eventually also migrate to mobile as productivity improvements make more efficient parallel hardware more pervasive. Developers tend to be researchers, with weak documentation and ad-hoc development processes. Almost all applications in this segment involve a model of some real-world phenomena. The applications are either simulations, or else solutions to models of large systems which directly give results that would otherwise be qualitatively identified from simulations.

Development Environment Tends to use Open Source tools, and a variety of languages, with Fortran a notable standout.

Distribution The software is downloaded from an Open Source site if it is distributed at all.

Many code-bases are internal to one group and never distributed.

Software Stack Open Source software stack, with specialized libraries such as LaPack and BLAS.

Driving Factors The problem being solved is the driving factor, with performance on a supercomputer paramount, in general. Reduced software effort is desired, and so is increased portability with high performance without hand-tuning. The software effort and portability seem to be gaining in priority but still appear to be far behind performance and verifiability of correctness.

2.2.2 Conclusions about the Segments

The most important pattern is the difference in needs and infrastructure between segments. A close inspection reveals that there may be room for more than one framework. A single segment will standardize on a given framework, but there is little or no desire to run applications created within one segment in the infrastructure of a different segment. So, each segment is free to pick the framework that suits it best, or the framework that is good enough and establishes itself first.[1]

2.3 Focus on Distribution and the Implications for BLISS

The distribution process is given special attention because it is intrinsic to one of the frameworks in part 2, called BLISS. As such, BLISS's distribution model may present a barrier to adoption for segments with incompatible distribution requirements. This section draws

[1] As will be seen in Chapter 4, uncertainty plays a significant role in adoption, so a major player in a segment can cause an inferior framework to be adopted just by backing it. The commitment to a thing, of a major entity, reduces uncertainty for all the others in that segment, which speeds adoption of that thing.

conclusions about BLISS's adoptability for each segment, based on the distribution processes listed in the previous section.

BLISS: The BLISS framework is organized around specialization servers. One of these collects application sources, then, to each, applies a number of specializations. Each specialization starts with the same source and ends with a different installation-bundle, targeted to a single type of hardware. The code that performs specialization is packaged into a separate module, one per hardware target. These modules are collected from many 3rd party sources, onto a server, then plugged in to the server's automated system.

Two separate distribution processes: Hence, in BLISS, there are two distributions in play: collection of specialization modules, with distribution to the specialization servers; and separately the collection of applications onto a specialization server, with distribution of installable images to end-users. Some of the software segments are more compatible with this approach than others.

Strongest constraint: The segment with the most constraining distribution process is shrink-wrap, where end-users obtain the software as part of the purchase process. The important characteristic is that distribution happens via recordable media, such as DVD, which is currently required.

Getting around constraint: For BLISS, each DVD can have a copy of the specialization server's output, for that one application, but that output resides on the DVD and cannot be updated with new specialization modules after it ships. Such update would require web access, forcing web-access to be a standard requirement for shrink-wrap software. This is becoming more common, but is not the current standard.

An additional complication for BLISS is the vendors' desire to control distribution. A consequence is that each vendor has their own specialization server, and so may have a different version of the BLISS standard formats than those a given specialization module was written to.

To address this, BLISS should include a process to minimize issues when the vendor's server-version differs from the version the specialization module was written for, and when both those differ from the version used at each stage of an application's development. Either a mix of the three should be tolerated, or else automation for modifying old applications to new standards implemented.

Segments that fit BLISS: For BLISS, the Open Source segment already follows the centralized server model in large part, so it doesn't present a barrier to adoption of BLISS. Likewise, the mobile application segment is compatible with a BLISS-style distribution system. That's because each service provider already has a central server that distributes applications for the mobile devices connected to their network.

Enterprise software is highly flexible because it is essentially many one-off custom software packages. However, access to the source code is restricted with vigor, so each application would require a separate specialization server set up just for it. Hence, problems may arise when trying to keep the specialization server up-to-date with the server that the specialization modules are written to be compatible with. Also, issues may arise with trusting specialization modules.

In the Embedded segment, the application developer requires control over the specialization server, which complicates things. It implies that only applications from one manufacturer would reside on a given server, but many such manufacturers may be involved in the value chain that produces the end product.

However, because these servers are not available to the general public, legal agreements can be put into place and trusted 4th party operators of the servers utilized, allowing all the

applications for a given product to be collected onto the same server and have the same specialization module applied to the full stack of applications. The fact of a very limited number of entities to distribute to makes this practical.

2.4 Model of the Development Process

In this section, attention is focused on the nature of development, because disturbance of existing processes creates a high barrier to adoption. Each framework should fit existing development work-flows and tools as much as possible, in order to minimize barriers that would slow the framework's adoption.

In addition, the programming language is used in the context of some sort of development process, even if it's just an ad-hoc process. New languages should be designed to fit with and support the processes dominant in the targeted segment and targeted team-size. Hence, development processes should be understood in more detail.

Toughest process to fit with: The most sophisticated processes are, arguably, in the safety-critical embedded sub-segment and the enterprise segment. Sophisticated processes are also used, across segments, by large consultant or contract development organizations.

2.4.1 Overview of Development Workflow

Figure 2.1 illustrates a typical workflow. It follows an iterative process, with a number of iterations shown. This makes a good example of an enterprise application being developed by a contract development organization. Each iteration shifts the effort among development activities, as depicted.

The figure shows requirements gathering at top, below that prototype development which generates use-cases that define what the application should do and how it should inter-

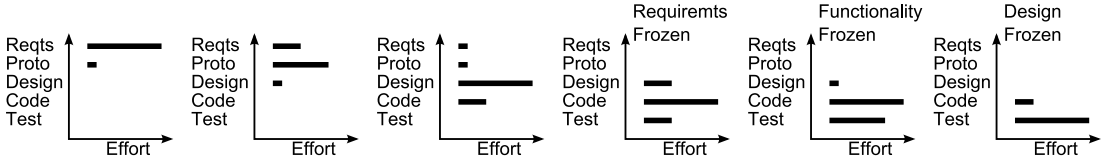

Figure 2.1: The areas of development activity, and shift among them from iteration to iteration

act with a user. Below that is design activity, which includes detailed specifications, software architecture, and module/object-model design. Below that is writing of code that implements the behavior of the use-cases and other requirements. At the bottom is testing, which includes unit testing of individual modules, as well as system-level (AKA integration) testing.

In the X axis of each iteration is "percent of an iteration's total hours of effort". This is repeated 6 times, each later in the project. In the first, the top has the most effort. As move to the right, the effort shifts down. The figures progress by week into the project. Only a few days elapse between the first two iterations, and longer between later iterations, which indicates that more accumulated effort goes into later iterations. Overall, more effort goes into implementation than requirements.

Freeze points: Markers indicate freeze points. The first represents a freeze of requirements. After this point, the user-level desires for the application don't change, The second freeze shown is for functionality actually experienced by a user. The last freezes the internal design and interfaces, after this point only debugging code changes are allowed, Final-release freeze is not shown, but it would come after an iteration where nearly all the effort is in system-wide testing. Different people are involved in each type of effort, so the people shift to different projects, or to later releases, as their skill-set becomes frozen-out of the current release.

Compatibility requirements: Whatever framework is adopted for the triple-challenge, it must be compatible with this iteration-driven evolution of software. In particular, the devel-

opment cycle of producing a bit of code, then testing it in isolation with some unit-tests that provide inputs to the piece and check outputs from it, and then system-testing (also known as integration testing) where the pieces work together and unexpected interactions manifest. If one had to compile to a wide variety of machines in every iteration between development and testing, that would slow productivity dramatically.

2.4.2 Details of the Development Iterations

The framework will have to be compatible with the process, which progresses in iterations, so the iteration details and progression are examined.

2.4.2.1 Early Iterations

The first few iterations are top-heavy with requirements gathering and specification-prototype creation. Requirements gathering involves interviews with users of the system, to discover what they want to see the software do, and how they want to interact with it. After the first interview, a specification-prototype is created by the requirements-gatherers. This is just a mock-up that shows the interface the user will experience, and a description of what the behavior will be like. Several short iterations cycle between user-interviews and developing the specification prototypes used to demonstrate during the interviews.

A given early iteration takes the previous round's interviews, generates ideas, and embodies those in a mock-up of what the user might see, and a description of how it would interact with them. These are often termed use-cases. The user examines the mock-ups and gives feedback, which prompts the next iteration of mock-ups, perhaps with some functionality actually implemented in the next round to back it up.

2.4.2.2 Middle Iterations

As the use-cases stabilize, they are handed off to designers, who create software architectures, object models, and so on. At this point, it is often discovered that technology constraints put force to modify the use-cases, which causes further iterations on those.

Aside, maintenance release: For a maintenance release, which happens after a software system has been completed, and just adds further functionality, the process has fewer iterations in requirements gathering, and less work in specifications/modelling. This is because the system architecture already exists and is only being enhanced, so less learning and discovery is involved. There are fewer degrees of freedom in the system that would cause lower levels to push back to higher levels. Also, the users already have a good idea of what they want.

Freezes: As the iterations stabilize, or pre-determined time-points are reached, freezes are declared, which signal that further iterations will have no activity in the higher-up portions. For example, the first freeze is on the use-cases. After a few iterations between interviews of users and mock-ups plus resource-constraints and other measurement constraints, such as "must take less than 1 second to respond" or "must require less than fifty thousand dollars of equipment", the requirements get frozen. Next, after design iterations interact with initial implementation, the design gets frozen. Later, once some unit testing and the first system testing has taken place, functionality will be frozen.

Pruning functionality: At each freeze point, decisions are typically made by developers about what they believe they have time to complete before the ship date. Functionality that will take too long is postponed to later releases of the software. It is only once implementation has been attempted that the true amount of effort required for particular function becomes known. Therefore, final functionality can only be set after reaching the implementation iterations. After

freeze of the use-cases, no feedback upward is possible, so the only alternative is to prune functionality out of the release, saving it for later.

2.4.2.3 Later Iterations

After the iteration that ends with the functionality freeze, the iterations focus on implementing that functionality, and the Quality Assurance team ramps up serious effort designing integration tests, building test-harnesses, and writing and performing test-cases. A series of implementation freezes happen, each of which isolates sections of code for testing.

The purpose is to give the Quality Assurance team sections of the system that are stable, in order to run integration tests on them. System testing requires long periods of time to run a series of tests that cause increasingly rare interactions among the parts of the system. These uncover interactions that give unwanted behavior. Each time a developer modifies code in a part of the system, all system-testing involving that code must be repeated. So, over time, portions of the system implementation get frozen, and no further development on it is allowed, in order to reduce the system tests that have to be run on the next build.

Finally, QA completes a system test that has only "minor" bugs found, and the build is frozen and released.

People shift to next release: As freezes take place, people shift over to higher portions of the next release. The people who work on the portions above the freeze line (above in the figure) move on to those portions of the following release.

2.5 Applying Development-Process Knowledge to Language Design

This section focuses on one proposed highly-productive language called WorkTable. It was designed with the development process for large teams in mind, and set in the context of BLISS. This section describes how features of WorkTable satisfy development-workflow needs.

Need to accommodate varying programmer skill levels: There is variation of programmer skill levels within large teams, such as in the Enterprise Segment, Shrink-Wrap Segment for large projects, and Embedded Segment large projects such as automobile control systems. For projects with such teams, a language that encourages decomposition by skill level and type will be a boon.

How Worktable addresses the need: The WorkTable language was designed for this case. It separates out the parallel portions of the code, leaving the bulk of code sequential. Only one or a few specialized developers create the parallel portions, and define sequential pieces that fit within the parallel scaffolding. They assign the sequential pieces to the rest of the programmers.

This works well for large projects from a workflow point of view. The language is designed to support a visual design tool that keeps the software architecture diagram synchronized with the code, allowing either to be modified and the result seen in both. A team-lead can work from the code-view and hand out pieces to team members, then modify the parallel portions when necessitated by structure discovered during implementation. The code change propagates backwards, showing up in the architects view.

The architecture-code integration increases the rate of design iterations, and enforces modularity, which increases the ability of later programmers to learn the code and modify it.

Parallel composability and isolation: The language is designed so that an application's modularity is maintained in the face of concurrency. A team lead is free to further decompose portions of code assigned to their team, exposing the parallelism structure and constraints on scheduling within it, without introducing unforseen concurrency bugs due to interactions with portions of code worked on by different teams.

This property is especially important for maintainability. Modifications to code have to be possible by examining only a modular portion of the code, without fear of introducing unforseen concurrency bugs. The fewer lines of code that have to be understood in order to make safe changes, the more maintainable the code base, and the lower the cost of maintaining it.

Chapter 3

Individual Human Factors

Having discussed the various segments of software, this dissertation turns now to the point of view of an individual programmer. It models some aspects of software related to personal preference. This addresses both productivity and adoptability.

This chapter starts from the perspective of the problems being solved. Developers will be most productive when the software directly captures how they naturally think about a problem. So the first section speaks to the productivity goal and suggests possibilities for language designs.

The second section talks about development process preferences that may have deep origins in a programmer's psyche. These speak to both adoptability and productivity.

The third section considers psychological aspects of people in a field, in regards to how those may affect productivity.

3.1 How People Handle Parallelism in the Real World

People deal with parallelism in the world around them every day. Event planning, project management, any place in our lives that we coordinate with other people working towards a common goal. This is, admittedly, more difficult than doing everything ones self, but it isn't as difficult as traditional parallel programming!

Therefore, looking closer at the ways people already handle parallelism may give some helpful clues to designing new, highly productive, parallel programming languages.

Duplicate scheduling decisions: Many approaches have been devised to handle real-world parallelism. The most common is for all the people involved to have the same scheduling algorithm, which allows each to predict what the others will do, based on having the same information. Driving a car is the most common example. All the people on the road are computing in parallel. It works because they all have the same set of traffic rules and all see the same inputs: signs, each other's car-positions and velocities, and road-boundaries. The car-driving analogy may make a good basis for a parallel language.

Separate scheduler: Another approach to handling parallelism in the real world is for one person to act as the scheduler that determines what tasks the other people receive. This is the role of the project manager. Each worker receives tasks from the manager, and reports when they're done.

Combining with duplicate scheduling: Workers are trained in how to interact with other workers when their tasks interact. A good example is building a house. The foreman assigns tasks such as laying foundation, tying down rebar, setting up forms, and so forth. They assign a certain number of people to each foundation, and those people then coordinate amongst

themselves, each starting in a different physical section. Such spontaneous organization happens without an explicit decision-maker because all have the same scheduling algorithm and same inputs, as in the traffic example. They even can see when one is lifting a piece of rebar and goes over to help placing it, and so on. This hierarchical scheduling model may also make a good basis for a parallel language. It will be seen in Chapters 7 and 8 to be a good model for specialization.

A process as a parallel application: A third example is designing a physical factory. An assembly line, chemical refinery, or other physical process like this is a parallel computation in the real world. People routinely design such things, indicating that the method of creating such designs may make a good basis for a parallel language. In fact, the WorkTable language briefly used as an example in the previous chapter follows this process-design analogy.

Math as a language inspiration: Finally, mathematicians handle parallelism by bounding the possible outcomes from it. Existential quantifiers forAll, thereExists, and so forth declare bounds that must exist. This is rather more of an inverse approach. Normally these are used in a theorem, which has a proof. The proof performs the steps of a computation that enforces those boundaries. The steps of the proof are often themselves inverse, with their own proofs that compute enforcement, and so on.

Mathematics is a bit odd in that way. It combines declarations of outcomes with computations that enforce a declaration by further declarations of smaller outcomes. At some points, the declarations must be enforced by the people who maintain the system of mathematics, for example the notion of a number exists only for people and must be enforced by people. This implied enforcement allows mathematics to imply computations that enforce mathematical declarations. For example, the declaration"1 + 1 = 2" relies upon people to perform the

computations to enforce it.

This is a subtle point, but important in the design of programming languages. Attempting to make a language that declares bounds on parallel behavior without an automated means to enforce the declared bounds forces the programmer to create the equivalent of a proof of those bounds. This is the case with the typical Thread constructs of mutexes, condition variables, and so forth, and is the reason those constructs are so difficult to use.

In the real world, people only handle parallelism with declarations of outcomes when they state *local* rules that each participant locally enforces for themselves. This is what traffic laws are. Examples of existential quantifiers are rarely seen. The equivalent of a mutex is only used in places where the natural world enforces it for us, such as "only one car may occupy this volume of space at a time". This should provide some good clues to how to design a high productivity parallel language.

3.2 Personal Preferences in Development

Each programmer has their personal preferences: some like mouse-driven visual development, others can't be torn away from command-line text-based tools; some work best all by themselves, others gravitate towards teams and interaction. These preferences affect productivity – force a GUI person to use text tools, and the lines per day is sure to drop – and vice versa! The reticence to adopt alternative approaches and persistent return to past preferred ones suggests that these may be deeply embedded in the person, as opposed to simply what they were exposed to first.

3.2.1 No One-size-fits-all Development

Industry factors often drive the type of development process and development tools used in a particular software segment. The result is that people migrate to work-places that have the development aspects they prefer. Hence, there looks to be no one-size-fits-all solution. The take away being that the framework should support many different languages and/or development tool approaches.

In summary, the aspects most relevant to personal preference include:

- visual development vs text-based tools
- strong process vs loose rapid development
- static typed language vs dynamically typed
- declarative language vs low-level control oriented
- team-based vs isolated development
- structured thinking vs creative flow

These personal preferences affect an individual programmer's productivity, and on teams, affect the emergent productivity of the team and organization.

3.2.2 Variation in Languages:

The language that is most productive for a one-off Open Source gadget is not likely to also be the most productive for a ten million line project with a thousand programmers. For example, a Perl script can be used to make a few-thousand-line widget to be released on SourceForge much faster than using the Rational process, with UML diagrams, formal uses-cases, and code documentation. But just try developing a 10 million line enterprise application with Perl scripts!

To a large extent, people filter themselves into jobs that have programming environments and processes that fit their personal preferences. Hence, it is unlikely that a single language, with accompanying development environment will be universally the best.

This suggests that several parallel languages will exist, for different software segments. Within a segment, several domain-specific languages might exist for different domains of problems being solved.

3.2.3 Conclusion, Framework Best when Supports Many Languages

What this means is that a framework should support many different languages and accompanying development tools and development processes and styles. The more constraints a framework places on language and development process, the fewer programmers it is able to supply high productivity to, and the less Adoptable it becomes.

As a consequence, with a wide variety of languages and development tools being supported, reuse of specialization tools and effort, among languages, becomes important for Adoptability.

3.3 Human Aspects of Productivity

The personal preferences for development affect individual people's productivity, within a given environment. However, this is mainly relevant in suggesting that no one language plus tool-set will fit all programmers. The conclusion is mainly that there is enough variation among people to expect a *variety of environments.*

This section explores the consequences of this, looking at the ways in which characteristics of individuals will affect the languages and development environments. It explores the consequences of the variations among people, which drives a variation in development process,

and variation in languages and tool-sets. It wraps up by drawing conclusions for the framework, which must support the variations.

Human Factors Relating to Framework: As will be seen in Chapters 7 and 8, on a model of parallel computation and a model of specialization,, the technical process of porting source code to run efficiently on target hardware requires the language to gather information from the programmer to feed to the porting process. Hence, the human factors discussed here will have to integrate with the portability factors discussed there. The framework design will affect how the two sets of factors can integrate, so it should be designed with models of both in mind.

3.3.1 Variation Among People

The wide variation in people involved in the software development process affects the language, tools, and process flow. For example, game design and web design includes graphic artists, scientific software development includes scientists, financial software development includes traders, and so on. The development processes include such people, with their variations in skill sets and levels of ability.

Tool suite integrates diverse people: Such variation will show up in the form of integrated suites of tools that together form a work-flow. The view of the world relevant to each person is presented to them to enter their portion of the project. For example, a drawing tool is presented to graphic artists to create drawings, but the object modelling programmer sees the drawings as objects with the name and purpose of the drawing attached. The tool, for the game-engine programmer just represents the drawing as an object of a given type that can be manipulated in the code. The frameworks will have to be compatible with such work-flows, supporting this style of development, while also providing portability of the game-engine code to a wide array of

hardware. But it will be best if the framework is not specific in any way to game development.

Small projects: For small projects, individual programmers will gravitate towards their personal preferences. So a variety of highly productive individual-developer oriented languages should be available. This allows the programmers to self-select, and covers projects that have no significant future maintenance requirements, allowing the language to sacrifice project-oriented features in favor of individual productivity features. One example is eliminating static typing in favor of dynamic typing.

Medium projects: For medium size projects and teams, a smaller number of more structured languages should be available. Such teams often spontaneously generate their own informal processes, using email, post-it notes, and mutually agreed-upon conventions.

Large projects: For large projects, the entity contracted or organizing the project will have a process. It will include significant documentation, a mature workflow, and other features that reduce risk, increase maintainability, and facilitate a large team working coherently. There is less room for individual preferences here. Instead, the focus is on being able to remove any programmer at any time and replace them and the project continues efficiently.

3.3.2 Match Language to Thought Patterns

In general, the closer the match between a person's natural way to think and the structure of the code, the higher their programming productivity. This suggests that many languages be produced, each of which matches closely to some domain of problems. The commonalities within the problems is directly captured in the language design.

Learning curve drives general purpose: A second factor is learning curve. Contract programmers have an economic incentive to cover many different domains. So, learning a new language that bears little in common with any others adds the learning time to the producing code time, lowering average productivity.

Barrier to non-expert customization: In addition, non-programmer domain experts may only want to customize some existing code. A large learning curve will be a barrier to them and also lower their average productivity. Further, if people are forced to specialize to a particular domain, due to high domain-language learning curve, their fees will increase. The cost of specialists is typically much higher than that for generalists.

Overall productivity lowered for short-time users: So, high learning curves reduce the overall productivity of programmers who spend only a short time using a domain-specific language. They also increase the cost of domain-specialist programmers, and cost is a primary driver to wanting higher productivity. So in all cases, high learning curve counters the productivity advantages of domain specific languages. Hence, learning curves must be kept low, which limits the degree of specificity to a particular domain, and so limits that aspect of their increased productivity.

Math-style language doesn't fit parallelism thinking: Most importantly, sequential languages have their origins in mathematical style thinking, which worked well because scheduling is inherently constrained to one possibility at a time. But this doesn't translate well to parallelism where many scheduling possibilities exist at a time, which requires proof-type thinking to guarantee correctness. People, in practice, think about real-world parallelism in ways different from that.

In particular, existential quantifiers like forAll and thereExists require proofs, which

are very difficult and unnatural in parallelism. Lock-based control of scheduling is also a poor match to the way people think about parallelism in the real world. So, highly productive parallel languages are not as likely to be based on mathematical-style mental models.

Suggests explore every-day handling parallelism: More investigation of every-day approaches to handling parallelism in the world around us is needed. New languages should be designed based on these already-successful and used-in-practice mental models.

3.3.3 Conclusions Based on the Complexity of Productivity

The previous sections have suggested that productivity is complex, because of the variation between personal preferences, the variation between skill sets, the variation between project sizes and development organizations, and the variation between domains of problems being solved.

No single language: This complexity, based on inescapable real-world factors, suggests that no single language nor single development approach will provide high productivity to the majority of programmers. Rather, a variation in languages and development tools will be required. To be productive, a language and tool-set must fit the relevant aspects of the real world, while also fitting the programmer's preferences and the project's needs.

Need rapid prototype of new language: One conclusion from this is the need for fast iterations while searching for highly productive language and development environment for a given context. The framework should have features that support rapidly creating a new language or development tool to test for fit to a particular set-of-developers + software-segment + problem-domain + organization + project-characteristics.

Possible reasons for past shortcoming: It may be that past shortcomings in productivity of parallel programming have stemmed from two factors: 1) the language design has been driven by technical factors rather than real-world factors, especially hardware and compiler details, and 2) history/culture has perpetuated early programming mental-model choices that were made when all programming was sequential.

Overcoming reasons, find highly productive language: To break out of such a local-minima in thinking, to solve productivity, several things may be helpful: 1) study how people think about parallelism in the real world, and make languages that have the same patterns; 2) identify categories of existing developer preferences and make at least one language for each category; 3) verify this dissertation's suggestions for software segments and make a set of languages plus tools for each segment; 4) identify categories of existing development processes and make at least one language plus set of development tools for each category; 5) identify categories of project-types and make at least one language plus development tool-set for each category; 6) pick domains of problems that share similar patterns and make at least one language for each domain.

The last 5 form a set of dimensions, however in practice they will only be sparsely populated. For example, problem domains will turn out to be correlated with programmer preferences, and software segment will be correlated with development processes, so the number of combinations will not be as large as it may at first seem. Also, many commonalities exist, allowing, for example, reuse of base language with variation only in parallelism constructs.

Things framework needs: The most important conclusion is that to achieve high productivity for the majority of programmers, the framework should support the languages and development tools for the majority of the combinations. And, to remain adoptable, it must also

be compatible with the development workflows and processes.

Commonality among languages and tools: For high productivity, commonality in mental model amongst languages and tools should be maximized, to reduce learning curve when moving between them. It is especially important to maximize commonality amongst domain-specific languages, finding a small number of common root languages and only adding variations for domain-specific patterns.

Identifying commonality in implementation will reduce the unique effort put into each variation, by maximizing reuse between languages and between development tools. The Adoptability of the framework will be improved by both, supporting the variation, and exploiting reuse within it.

Chapter 4

Model of Adoption

Why adoption is important to model: Adoption is one of the two most important drivers of frameworks such as those proposed in part 2. The main reason that reasearchers, and later people in industry, will choose one particular framework over another is the belief that the research they contribute is more likely to be widely adopted when it's done in terms of that framework. In other words, the question is: "I'm a researcher on some aspect of parallelism. Why should I do my research within the framework proposed in part 2?" The answer is "You don't have to re-create a framework for yourself, saving you work, and saving you time, and lowering the barrier to trying out your ideas. And, your work is more likely to be adopted when done within the framework, because the people who designed the framework did all the effort of understanding the software segments and how such infrastructure is adopted – the pushes, pulls, barriers, and uncertainties. So, you don't have to think about any of those – just fit into the framework, and it takes care of aligning with adoption."

Adoption is equally as complex as productivity, influenced by human psychology, existing business practices, economic drivers, personal preference, and so on. Adoptability overlaps with productivity in many ways.

Chapter preview: This chapter will overlap with the previous in many real-world areas, but with a sharper focus on Adoptability as opposed to productivity. It will also overlap with Chapter 2's coverage of software segments and their practices, but again with a focus on Adoptability. It will expand detail of industry structure and practice, mainly on how decisions are made, the priorities driving them, and any fixed features that affect the decisions.

Adoptability is essentially about decision making, the decision to choose one method over another, or one tool over another. When it comes to parallel programming, as the previous chapters have shown, many factors are involved in the decision of which language and accompanying development tools a given programmer uses.

Model of Change: Change, to adopt something new, boils down to three things: push, pull, and barrier. Those are modulated by degree of certainty in them. Pain that a person or entity is experiencing pushes them to change, while promise of satisfying desires pulls them to change, and the cost of transition from current to new is a barrier blocking change. Each of those has a degree of uncertainty, and the final decision of whether to change is a balance between expected amount of benefit vs cost, each multiplied by the degree of certainty that benefit will materialize and the cost will not be higher. The decision to adopt a new language or development practice follows this pattern.

In particular, the economic cost and mental pain of current parallel programming approaches is pushing, strongly, for a satisfactory solution to the three goals. This differs from the case of sequential programming, where things are relatively stable in the various segments. There is no large pain driving change in sequential programming, while the cost of changing is high. Hence, low expected benefit, with high cost, means low chance of adopting a new sequential language. In contrast, for parallel languages, the pain pushing for change is ever mounting, and increasingly severe. This opens the opportunity for disruptions like adding a

new form of software infrastructure, such as the frameworks proposed in part 2.

Size matters: The way the decision to adopt change is made is driven by the size of the project in combination with the size of the entity incurring the benefit and cost of the project. This is because the tolerance for uncertainty correlates with size of project plus size of entity, and drive to change varies according to size of project and entity.

One person projects have the most freedom over language and tools, and is where the most experimentation takes place, especially in the Open Source segment. People who have experimented with various languages on their own band together into groups for medium-sized projects, and collectively choose the language and tool-set. The fact of being a group constrains the choice of language in many ways, as discussed shortly. Finally, large projects are performed by entities that have done many projects in the past, with established processes that have evolved over time. These have the most constraints on which language and tool-set is used, and the lowest tolerance for uncertainty, which implies risk.

4.1 One Person Projects

Few constraints on choosing: One-person projects enjoy the most freedom in choosing language and tool-set. The greatest is in personal projects and the Open Source segment where the motivation is learning, making a tool to solve ones own problem, or the fun of seeing what one can accomplish,

Influences driving choice: The decision to adopt is influenced by: recommendations of others, potential for future employment, novelty or curiosity, peer-pressure, trendiness (social momentum), objective evaluation of features, commercial vs Open Source, opportunity to contribute, maturity of the language, availability of supporting development tools, availability of

tutorials/training, learning curve, applicability to a domain of interest (used as part of some other activity, like making a game, robotics, aso), level of productivity for problems of interest, breadth of applicability (learn one language, use on all problems interested in), feeling of power when using it, fit to natural way of thinking, and fit to preferred development style.

When examining these factors, some are derivative from a "critical mass" of others (trendiness, peer-pressure, employment opportunity are derived). Some are due to appearing in the opportune place at the opportune moment (trendiness, peer-pressure). others are due to backing by a trusted entity (trendiness, peer-pressure).

Large number of languages used: As a result of the large number of factors, and the large variation among people, there exists a large number of languages used in one-person projects. This is where experimentation with language design takes place, and adoption of change is easiest. Due to the human factors discussed in the previous chapter, these projects have a "long tail" phenomenon for language usage.

4.2 Medium Sized Projects

For medium sized projects, decisions are made on a team basis, with 2 to 10 people in a team. There is less tolerance for uncertainty about benefits and costs of something new, compared to a one person project, but much greater tolerance than for large projects. In addition, team development aspects come into play, as do revision aspects of development.

Hence, this is where new project-oriented languages and tools are generally first adopted. The teams most likely to adopt change are skunk-work projects in large corporations, and teams in smaller companies. Skunk-works teams are non-critical projects driven by technologists who appreciate the technical aspects and want to use a new approach for personal desire. They

often grow organically. Small company teams are free from uniformity constraints that arise in larger companies (due to development cost, risk, training cost, and management-structure reasons), and are pulled by the promise of advantage over others. The desire pulling is strong and risk-avoidance is low, so experiments with new languages is high, in comparison to large entities.

4.3 Large Projects

Large projects have history: Large projects are never first-time projects for an organization. They are performed by an established entity that once started with small and medium sized projects, then matured as it grew to ever larger projects.

Slow change: Between projects, or revisions, the development process changes only slightly, and evolves slowly over time. At each stage in the growth of the organization, changes to development are driven by pain felt during previous development and slowed by momentum of the existing development process.

As a result, adoption decisions are localized and highly constrained. There is a large base of programmers already trained in the existing process, and a large expense in training, and cost of disruption to development caused by large changes in process. So, changes tend to be incremental.

Risk avoidance: Large entity decision making tends to put more weight on risk avoidance, for many reasons that are out of scope. So they favor stability of the workflow and development process, predictability of development cost based on past experience with the existing process, continuity of development operations without a disruption to train on the new approach.

Conditions for change: A discontinuous change in language and development process only happens at larger entities when they have a compelling pain and a low-risk solution is available. The risk of adoption is set by the size of disruption, which sets the cost to adopt and degrees of freedom for things to go wrong. This is the barrier against adoption. In the other direction, pain experienced during previous projects and threat represented by competitors gaining an advantage with a new approach are drivers towards adoption.

Adoption modelled as dynamic system: In general, adoption in large entities must be modelled as a dynamic system. It was, for one-person projects, a project-by-project decision, fresh each time. But for large entities, adoption of a language plus tool-set is a long process involving many independent decision-making entities, and so evolves, as a dynamic system does, over time.

Discontinuous jump to new language: A discontinuous jump to a new language and tool-set generally happens in one of three ways: 1) the entity purchases/merges with a different one with a different development process, 2) high pain drives the change, and a low-risk solution is proven by an outside entity, 3) high pain drives the change, and a low-risk solution has organically grown inside the entity.

Adoption of large change can be driven by acquisitions and mergers that introduce new development practices into a large entity. This type of adoption of change is driven by factors unrelated to the language or development process.

Adoption of large change can also happen when it has been soundly demonstrated at other large organizations to solve the same pain. The closer to the same domain, same development style, and direct competition of the other organization using the new language, the stronger the push to adopt it. The stronger the push to adopt, the higher the cost barrier can

be overcome.

Thirdly, adoption of large change can happen when non-critical internal projects demonstrate the benefits and certainty of the new approach. One-person projects happen inside medium-sized and large organizations. In a large organization, small teams driven by technical decision makers often disregard the business reasons for staying with previous development practices. They will ignore the business risk in favor of reducing the technical risk, and adopt a new language or development process for a small, "low profile" project. When successful, they often move up to larger projects, and so can grow a new language and development process from within. The success of such an approach depends upon the culture of the organization. Ones with higher risk avoidance and ones with stronger alpha-male characteristics tend to squash the new approach before it grows.

Adoption spreading across a segment: Across a software segment, the pace of change can be low when the barrier is high or pain driving change is low. At first, no entities in the segment have adopted, so all assign high uncertainty to the benefits and costs because any proof-of-solution entities are too dissimilar. As a result, the barrier, and momentum of existing practices, will not be overcome and no adoption will take place. Even if the reaction is net positive, the non-linearity of the decision making may mean it never takes place at all for that segment.

Chapter 5

Overview of Portability

Portability has the most well-defined technical depth of the three goals, and requires three chapters to cover a model of it. The first part of the portability model is a basic computation model, which is then used to analyze parallel computation. The basic computation model is in the next chapter, and the model of parallel computation built on top of it is in the chapter after that. The third chapter, Chapter 8, then uses the first two to identify what is needed for high efficiency portability. This chapter overviews the next three, as a first iteration in conveying the portability model and requirements.

Portability is defined as a single source running with high efficiency on many target hardware platforms. The wider the array of targets, the higher the portability. The closer to maximum performance on a given target, the higher the quality of the port to that target.

5.0.1 Specialization

Specialization is defined for this dissertation as the subset of porting activities that relate to performance on the target machine. The act of porting source to a target involves many things, such as translation from one set of OS calls to another, translation from the

source language to machine language, data-type changes (32bit vs 64bit), alignment changes, data-layout changes, static scheduling changes, and so on. The portions that relate to the performance on the target are collected under the term specialization. The code is modified in a different way for each target.

Hence, the quality of a port is determined by the specialization process for that target. Specialization is the heart of portability, and makes up the majority of the porting process when high efficiency on the target is desired. As a result, the terms specialization and porting might be used interchangeably.

The goals of the next three chapters are, to gain insight into how to design languages that feed the specialization process what it needs, and to learn how to design the best possible specializer for particular hardware.

Achieving that requires a model of parallel computation that informs about specialization. Specialization involves real applications and actual hardware, so the model has to include features of applications and actual hardware, which makes it a fairly high-level model. It will need some compatible low-level, basic, model of computation as a basis.

5.0.2 Why a New Basic Model of Computation?

What should the basic model of computation look like, given that it's the basis for the model of parallel computation? Can an existing one be used? Evidence suggests that scheduling is at the heart of parallelism, so the high-level model should revolve around the scheduling process. In turn, the basic model of computation should feature scheduling as a fundamental component.

As it turns out, the scheduling process is not a fundamental feature of previous basic computation models such as the Turing model [69] or lambda calculus [21] or PRAM [29]. This motivates the simple, basic model of computation given in the next chapter, which features

scheduling as major primitive component. It will serve as the prototype for the higher-level model of parallel computation given in the following chapter, which will be used to deduce the needs of specialization in the chapter after that.

5.1 Overview of the Basic Computation Model

The purpose of the computation model is to serve as the base for the higher-level parallelism model, which informs specialization. The primary requirement is that it feature scheduling prominently, in order to conveniently support the parallel computation model.

The model treats everything as a processor. For example, data doesn't exist outside of a processor.

A processor has three active elements: a namespaces of other processors, a pattern-manipulator and a scheduler.

Interestingly, any one of these elements can be modelled as implemented by a processor, which in turn contains the element being modelled. This self-recursion is a good sign. The lowest-level definition of processor should not rest upon even lower-level primitives! It is expected that the lowest-level model of computation is self-recursive, thereby avoiding the inconsistency of being in terms of things even lower level.

This self-recursion also provides the key to defining things like time, virtualization, execution model, and semantics.

5.2 Overview of Parallelism Model

The higher-level model is termed the Holistic Model of parallel computation. Its purpose is to model real-world machines running real applications in a way that tells what needs to be done to specialize particular software to any one of a variety of target parallel hardware.

To achieve this, it has to be generic enough to cover all or nearly all expected machines that will be built, exposing common aspects that affect performance. It would be helpful if it also indicated why past approaches failed to achieve wide and/or high quality porting. It should suggest ways to make future languages better for portability and be detailed enough to suggest techniques to specialize particular applications to particular machines.

5.2.1 The Scheduling Loop

The model centers around scheduling, which is inside a loop. First in the loop is a scheduling decision, which chooses a task and assigns it to resources. The second step is communication of the chosen task plus its data to the physical resource. The third step is computation of the task. And fourth is communication of completion status back to the scheduling process. Then it repeats.

This is idealized, abstracting away details about location of the scheduling decision processing, the fact of data for a task often being gotten as-needed, and the fact that scheduling can be distributed, requiring more than just completion status, and having non-uniform communication times of status to each distributed element of the scheduler. It also ignores the hierarchical nature of parallel computation, which implies scheduling at multiple levels.

5.2.2 Conclusions Drawn from the Holistic Model

Despite these simplifications, the model exposes important properties: applications have a set of characteristics which affect the effectiveness of scheduling; several classifications of scheduler exist; particular hardware characteristics influence which scheduler class is best for particular application characteristics on particular hardware.

This means that an application can be statically analyzed, to determine its characteristics. Separately, each hardware target can be analyzed, to determine its hardware characteristics.

Once the combination is known, the application characteristics can be used to determine what kind of scheduler is best for each phase of its computation, on the given hardware.

If the target hardware is known beforehand, and the phases of the application are known, then a *static* specializer in the toolchain can pick the optimum scheduler type for each portion of the application.

The specializer will need to determine in some way what code is in each task-type in the application, and what the characteristics of that task-type are. For the hardware it will need to know details such as memory hierarchy, communication characteristics, and processor speeds.

The model identifies the application characteristics most important to scheduling, which are: stability of task execution times; prediction ability of the execution time for a given task; minimum and maximum limits on task-sizes; ability to resize a task; prediction ability of the data footprint of a task; variability of dependency pattern among tasks; prediction ability of dependency pattern.

For example, scheduling can be done statically, in the toolchain, but it only results in high efficiency for applications with a particular combination of properties: the data footprint is predictable, the task size fits the machine or can be resized (statically) to fit, the tasks have no variability in execution time or else all execution times are statically predictable, the dependency pattern is regular, stable, and predictable statically.

A predictable data footprint is needed in order to pick assignments to processing elements which minimize the movement of data. Tasks that produce should be as near as possible to tasks that consume. The task size has to fit to the memory hierarchy, and its computational-complexity-induced communication must be minimized (consider matrix-multiply: total communication volume goes up as task size goes down). With a fixed, static, assignment of tasks to processors, idleness can only be avoided if the execution time, of a task, is known statically.

Finally, by definition, static scheduling is only correct if the dependency pattern is known statically, which requires it to be stable and predictable, and it has to be fairly regular to make the computation of static scheduling practical.

The other types of scheduler and more depth on the effects of hardware will be covered in the chapters on the Holistic Model and Specialization.

5.2.3 Main Result of Holistic Model

The main result of the model is the tradeoff between scheduler think-time and consequent amount of total computation time. Some application-hardware combinations show good performance with very simplistic schedulers that think for only a short time, others have highly non-linear landscapes requiring significant scheduler effort in order to achieve reasonable execution time. In addition, the size of communication time of status information shifts the parameters of the think-time vs work-time relationship.

As a result, the total completion time is characterized by the application's characteristics as they relate to scheduling decisions, the nature of the scheduling process, and the hardware's characteristics as they modulate the relationship between application and scheduling process. The model addresses this by suggesting categories of application characteristics, categories of scheduling process, and how the hardware characteristics influence the best scheduler for each application category.

The model is informal. As with everything in part 1, it has not been published. However, predictions from the model have so far proven consistent with informal observations.

5.3 Overview of Specialization

As mentioned, specialization is the process of modifying code to run highly efficiently on target hardware. The Holistic Model indicates that the heart of specialization is scheduling.

In more detail, specialization does both static and dynamic scheduling. The transition point depends on application characteristics modulated by hardware characteristics. However, for most hardware, task-code will be chosen statically (with the possible exception of re-choosing with a run-time binary optimizer). For heterogeneous hardware, many of the tasks will have multiple versions of their code generated statically, one version for each type of core. When multiple versions exist, some form of dynamic scheduling will choose one for each task at run-time, for example, choose between running a task on a CPU core vs the GPU.

How such a runtime scheduler decides which type of core to run on is more complicated, involving tracking location of input data, communication time to transfer between caches vs onto GPU memory, and the expected running time of the task and expected size of data to transfer. Designing and implementing such schedulers is a research focus area, that the frameworks in part 2 are designed to support and coordinate.

Chapter 6

A Basic Computation Model Featuring Scheduling

This chapter presents a model of computation that was constructed as an aid to understanding parallelism and has guided the work described in this dissertation. The model has been defined in precise detail elsewhere [42], and is used here only as an aid to understanding. Therefore, it will only be covered with enough depth to be useful as a thought-aid for the rest of the dissertation.

The model has the useful property that its basic elements can be easily identified within many things: within a program, within the language it's written in terms of, as well as within a schematic of a circuit, and even in the physical arrangement of atoms that make up the physical circuit. This flexibility allows it to inform the design of languages in general, as well as the design of circuits.[1]

[1] In contrast, Lambda Calculus and Turing Machines have been useful for theoretical studies, but not so much for language design. Lambda Calculus has been useful mainly in the design of Functional languages, and Turing Machines perhaps for the design of Finite State Machines but neither has provided insight into optimization, nor the design of parallel languages. Likewise, Pi-Calculus and other existing parallel computation models have limited scope of insight, all of which helped motivate this dissertation's model.

In this model, everything in existence is viewed as a processor, or part of one, and in particular, all data exists inside a processor.[2] Each processor is created according to a specification.

This chapter first states the elements of a processor, then describes how they work together to accomplish computation. Then, it defines a notion of time, as being the progress of a computation, and introduces the notion of animation as the means by which that time is advanced.

It leaves the exact details of the elements vague, because they are instantiated from the context the model is applied to. What remains constant is the presence of three active elements, with their relative functions, and the fact that data remains inside a processor at all times.

Examples of applying the model, and precising the details of the elements, are given throughout the chapter. For example, it illustrates how common programming languages can be viewed in terms of the model, and how patterns in code correspond to elements of the model.

With this as a basis, the following chapter goes on to apply the basic model in a complex context, that of parallel computation. The last chapter of part 1, Chapter 8, then applies the parallel computation model to specializing parallel code for parallel hardware.

6.1 Definition of Processor

In the model, everything is viewed as a processor, where a processor has three active elements: a namespace, a pattern matcher-and-manipulator, and a scheduler. It also has state, called the working-state. This is illustrated in Figure 6.1

[2] A distinction will be made between Data and Pattern later in this chapter, once enough structure has been presented to use within the explanation.

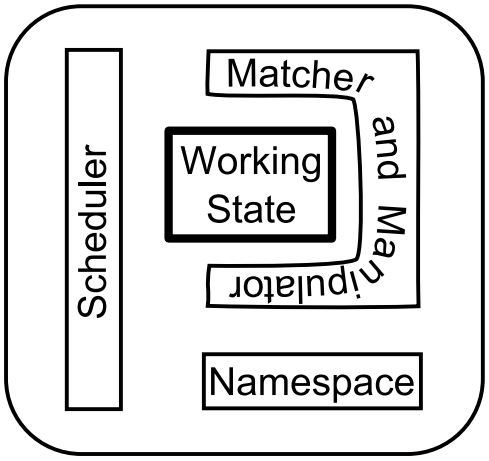

Figure 6.1: Internal elements of a processor.

The namespace attaches names to other processors and provides the means to communicate with them. It takes data as the "name" and a second bit of data to deliver to the "named" processor. In this model, all communication between processors takes place via a namespace. Some namespaces can also take the name of a specification instead of the name of a processor. These create a processor from the specification, then send to that processor.

Relating this to common terms, a network would be generally considered a namespace. And, a C style switch statement, or a dispatch pattern, is considered an implementation of a namespace.

The pattern matcher-and-manipulator recognizes patterns, and can remove a pattern, including its inputs. It can also fill the hole left, with a different pattern, usually a "simpler" one sent back as return.

The scheduler examines the patterns, identified by the matcher, and checks the constraints on them. It chooses from the ones that satisfy all constraints, and chooses a name to send it to. That pattern is removed, by the matcher-and-manipulator, and given to the namespace, which

sends it to the named processor (removing the pattern and handing it to the namespace turns it into data, as defined in Subsection 6.2.5). The receiving processor receives the data according to the pattern specified in its interface.

The working-state holds patterns, which change as the computation goes forward. Each particular value of the state represents one point in the computation. The working-state is normally set during creation, to patterns stated in the specification. These determine the behavior of the processor.[3]

The working-state becomes modified in three ways: inputs are inserted into the patterns in the working-state, coming in from an enclosing namespace; the scheduler can cause patterns to be removed; and removed patterns can then be replaced by a response sent as return. Changes in the working state define the progression of the computation.

The difference between data and pattern is defined in Section 6.2.5, once enough concepts have been given. However, both exist inside a processor at all points. They are taken from working-state and handed to the namespace, which transports only data. The receiving processor can turn the data back into patterns (or not), and place it the into its own working-state.

Activity Within a Processor Figure 6.2 The working-state is at the center of activity in a processor. A processor is created with working-state set to a pattern taken from the specification the processor was created from. Normally the pattern has holes in input-positions and so no sub-patterns are free to be scheduled.

To start things going, an enclosing namespace delivers data, which is cast into a pattern at the interface. The pieces of the pattern are placed into the working-state, in positions stated by the processor specification or fixed by the creator of the processor. These insertions typically flip scheduling constraints from unsatisfied to satisfied. The matcher-and-manipulator identifies

[3]Actually, the patterns determine part of the behavior, the details of the three elements determine the rest.

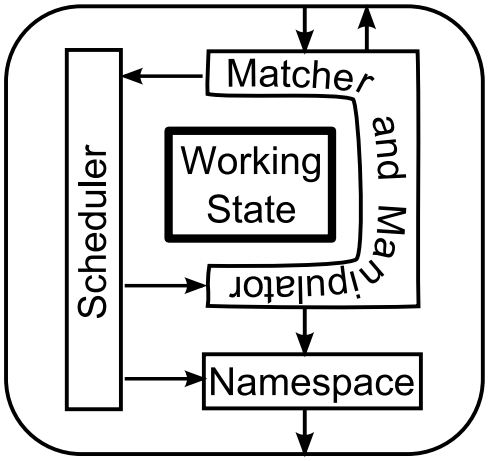

Figure 6.2: Activity among internal elements of a processor.

all the sub-patterns within the working-state and tells the scheduler about them. The scheduler checks constraints on identified patterns, and chooses from among the free ones. It then chooses the name(s) of a processor(s) to receive it (them).[4]

The scheduled patterns are removed, by the matcher-and-manipulator, along with whatever data is in their input positions[5]. This is handed to the namespace, which sends it to the receiving processor. The pattern itself determines which processor(s) the data may be sent to. The scheduler chooses the name of a particular processor instance, then the namespace sends the input data, arranged according to the pattern, to the named processor.[6] [7]

Returning a Result: A receiving processor may send back a replacement for the pattern it was sent. The replacement is inserted by the matcher-and-manipulator into the hole left by removing the sent pattern. Such insertions cause unsatisfied constraints to flip to satisfied. The

[4]Constraints include properties of the processor a given pattern can be scheduled onto.

[5]This provides one distinction between data and pattern: whatever is in input-positions of a pattern is data. The fact of being in the input position makes it *data*.

[6]For many languages, the processor is created just before the inputs are sent to it. In this case, the scheduler names the specification to create the processor from.

[7]Only data is sent through a namespace, but it may be arranged in a particular pattern that matches an interface pattern of the receiving processor. For example, in a language like C, the pattern is that argument data is ordered.

scheduler then chooses from among the free patterns and so on, until the working-state is empty, or no possibility exists for more patterns to become schedulable.

How a given processor decides to return a response depends on the processor specification (or creator). Often, when no more schedulable patterns exist, whatever is left in the working-state is sent back as the response.

Example: Take a simple example from a keyword-based language, like C. A pattern consists of a keyword plus its input-positions. The input boundaries are determined, normally, by syntax plus grammar rules, and a given input consists of all the syntax within an input position. For example, in "A + (B - C)" the + keyword has A and (B - C) as its inputs. The (B - C) is a compound input. In most sequential languages, this makes the + unschedulable. due to scheduling constraints blocking it. But the pattern with the keyword "-" in (B - C) has inputs that satisfy constraints, so it is scheduled.[8]

Relating the model to physical circuits: A circuit element can be viewed as a processor that has portions of its working state fixed, in the form of arrangements of atoms, and also has portions that are variable. The amount of field in a field-effect transistor's gate volume would represent variable working-state. The wires act as namespaces, while fields (voltages) moving down the wires are data in transit in the namespaces. More on this in Section 6.1.2.

Two Types of Processor: Relating the model to circuits shows the existence of two distinct types of processor: consumable vs persistent. Consumable are created with working-state holding a set of patterns, which are then consumed as the computation moves forward, reducing down to the final state. This is returned via the namespace that brought the input.[9]

[8] For sequential languages, only a single pattern is ever free to be scheduled.
[9] They may also communicate a result to a separate memory-processor, which is what's known as a side-effect.

In contrast, persistent processors are created with fixed pattern, in portions of working-state. These don't change during progression of computation. The fixed-patterns cause changes in the incoming data, which are received into variable portions of the working-state. The processors sends the changed data further on. The equivalent of returning a value happens by creating a loop in the namespaces, such as with wires.

Most programming languages, such as C, Fortran, and Lisp, specify consumable processors.e Dataflow languages specify a hybrid, in which persistent processors create consumable ones, when inputs are ready. Circuit schematics, though, specify persistent processors.[10] An OS-created process and a Thread are also examples of persistent type processors.

6.1.1 Memory Processors

In practise, memories are made distinct from processors, for example DRAMs and caches, vs CPUs in physical machines. However, this distinction is not fundamental, but rather one of convenience. Memories can be viewed as collections of normal processors.

To see this, consider that a processor can be created with working-state already containing patterns. A basic memory processor, then, consists of two processors, one that receives the read vs write pattern, and the other that holds the stored state. The one that holds the stored state is created with the stored pattern in its working-state.[11] A write causes the old state-holder to dissipate and creates a new processor with the new contents, attaching it to the same (and only) name in the read-vs-write "gateway" processor.[12]

[10] Although the story is not simple: for example, dynamic logic can be viewed as creating a virtual processor during one half-cycle, which is consumed during the other half-cycle.

[11] an alternative definition motivated by cross-coupled flip-flop circuits is that a pair of processors continually send whatever they receive to the other. A write interrupts the cycle, sending a new pattern, while a read temporarily adds itself to the name of one of the two, so it receives a duplicate.

[12] The reason the lowest-level processor can't receive both a write and a read pattern is that the basic processor model was chosen to have only a single working-state, which the received pattern is placed into – preventing the bottom-most from having both a fixed pattern (the contents) and choosing among commands (read vs write). The higher-level dispatches read to the lowest, and write to the creator in the namespace, while the lowest just returns its working-state.

A memory array, such as a register set or a DRAM bank, adds a third level of processor. The namespace in this top-level treats an address as a name and attaches it to one of the basic memory-processor combinations. The top-level takes an address as input, and hands that to its namespace as the name.

A constant in a language can be viewed as data in the specification, which is placed into working state during creation. Or, it can be viewed as a basic memory-processor pair. It is created with the constant value in the working state of the "storage" processor of the pair. The gateway processor simply doesn't have the write pattern in its interface. A constant could also be viewed as simply one of the storage processors. The interface only has one pattern, and receipt of it causes the input to be discarded and the scheduler to send back the working-state contents.

6.1.2 Applying the Model, Take One

At this point, the above concepts are illustrated with a quick demonstration of mapping the model onto common things in software development. More concepts will be introduced in upcoming sections, then the chapter will conclude with a second look at applying the model,, covering all the concepts.

Specification of a Processor: For C, a function definition is viewed as the specification of a processor. The function-name is the name of the specification. The code of the body is inserted into the processor working-state during creation,[13] and the parameter-names are used to identify where in the working-state to insert inputs.

[13] Actually, the body-code is converted into patterns that are in a format that the matcher and scheduler can manipulate.

Things Implied by the Language: One thing a language implies is scheduling constraints, attached to syntax and grammar. In C, scheduling constraints include the rules of precedence, fact of call-by-value, the semi-colon, and so on (because each of these constrain which keywords can be scheduled). A language also implies the specification of a *primitive processor* for each keyword and operator. And it implies the precise definition of how each internal element in a processor works. For example, the scheduler for a C processor does not work the same as that for a Prolog one.[14]

Creating a Processor: A function call causes the creation of a processor. It is specified to do this: when the call's constraints are satisfied, the scheduler hands the name of the called function to the namespace, along with what is in the parameter positions. The namespace looks up the specification (function body) and creates a processor from it, then passes what was in the parameter positions to it.

Studying operational semantics gives an indication of how the patterns in the working-space can be modelled. It also indicates how the patterns are consumed. A duality exists here between the consumable processor that models the behavior of the language, and the circuit-like behavior of machine instructions. This will be resolved after looking at the concept of animation in the next section.

The language implies details of the creator and therefore of the processors it creates. It also implies the way the namespace associates the function-name to its attached specification (the function body). [15]

A scheduled pattern is removed from the working-state, along with whatever patterns are in its input positions. The pattern itself determines which processor(s) it may be sent to.

[14]This example shows that the model is purposely vague, taking exact details from the context in which it is applied.

[15]For example, C takes the closest specification with that name that appears above the call in the same file. The "#include" directive is used to cause other files to be inserted, for this reason.

For a language like C, the pattern has either a keyword or an operator in it, which is the name of the type of language-primitive processor to send the input-positions of the pattern to. Or else the pattern is a function-call, where the function-name is the name of the specification to use to create a processor to send the input-positions to.

Note that a processor has an interface that defines the patterns it can accept from the processor-namespace. This plus scheduling constraints are related to the notions of syntax and grammar.

6.1.3 Hierarchy of Processors Contained Inside Others

This subsection explores the nature of namespaces, and the meaning of one processor being in the namespace of another. When one processor is attached to a name in a second processor's namespace, the first is defined to be contained in the second. An important point is that contained processors are not visible outside of the parent's namespace.

Consider Figure 6.3, which shows a hierarchy of processors. The children of a node are the processors attached to names in the parent, and so are contained in the parent. This figure shows a tree, but in general, any graph is possible. A processor may even contain itself in its own namespace.

In the tree, none of the children is visible outside of the parent. This means that a parent is not able to see grand-children nor any lower descendants. In other words, from outside a given processor, only its interface can be seen, and all processors in its namespace are invisible from outside that interface.

This is used for benefits in software and hardware. For example, polymorphism relies upon this: it presents a fixed interface, which accepts multiple commands. Each command is the name of a processor-specification in the internal namespace. Polymorphism works by changing the specification of the processor attached to a given name.

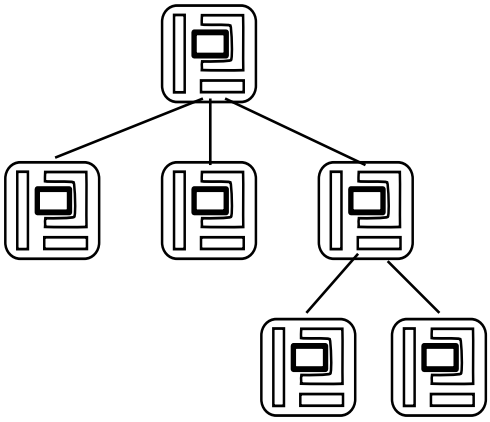

Figure 6.3: A tree of processors. The children are "inside", or contained in the parent processor.

In hardware, both a single-function-unit pipeline and a multiple-function-unit pipeline run the same executable. Each function-unit is a separate processor, which is possible because they're not visible from the interface! The Instruction Set defines the interface.

6.2 State of a Processor, Time, and Animation

In this section, a set of subtle topics is tackled, which involve the nature of time, and how it advances. Time is defined in this dissertation as the progression of changes in state. So this section begins by pointing out the types of state in a processor, and then relates those to the time in which they advance.

6.2.1 Kinds of State in a Processor

A processor has two different kinds of state: its working-state, and the hidden internal-state of its elements.

The working-state is initialized with the patterns that determine the processor's behavior. It then receives inputs coming in from its containing namespaces, and receives returns coming back from contained processors. The particular state-value represents the progression

of the computation of the processor. Hence, each change of this working state equates to one step of the processor's time.

From the point of view of a processor, its elements simply "work". Their actions take place outside of its time, which implies they have no state visible to the processor.

However, their actions must be performed! Their behavior has structure, which implies state for intermediate points of the behavior, which implies a separate time in which that state evolves. This gives rise to the concept of animation.

6.2.2 Animation

Animation is a separate time-line in which the internal state of the elements of a processor is evolved. Animation is performed by a different processor, which loads its working state with the internal state of the animatee.

Figure 6.4 illustrates this.

The animator, seen below, is a persistent-type processor (which is typical for animators). It is animating the processor seen above it. This animator has fixed working-state patterns that encode the behavior of the elements of the animatee. The animator loads the variable portions of its working-state with the element-state of the animatee. As the animatee's element-state progresses, this progression causes changes in the working-state of the animatee.

Many changes of animator state may take place to a accomplish a single change of animatee state.

This is the essential act of animation: progressing the changes in the hidden internal state of the elements causes progression in the working-state. In other words, advancement of the working-state of the animator in turn causes advancement of the working-state of the animatee.

Some readers may ask the question: what animates the animator? The answer, so far,

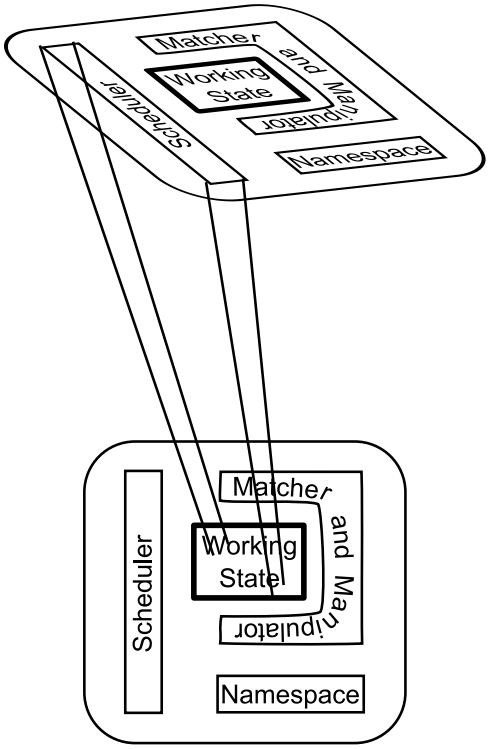

Figure 6.4: A lower-level processor animating a higher-level one. This shows a snapshot where just the scheduler element of the animatee is loaded into the working-state of the animator.

is that a chain of software-processor animators exists, and the lowest of these is animated by the physical CPU, which is in turn animated by fields moving in its wires, which are animated by quantum mechanics. Where it stops is an open physics question.

Figure 6.5 shows the ladder of animators.

6.2.3 Animation vs Namespace

Having seen both one processor animating another, and one processor being contained inside another, the question arises: what, exactly is the difference? Figure 6.6 illustrates the difference.

In the tree, all the children of a parent-node are contained inside the namespace of the parent. But the animating processor below the tree never appears in the namespace of any of

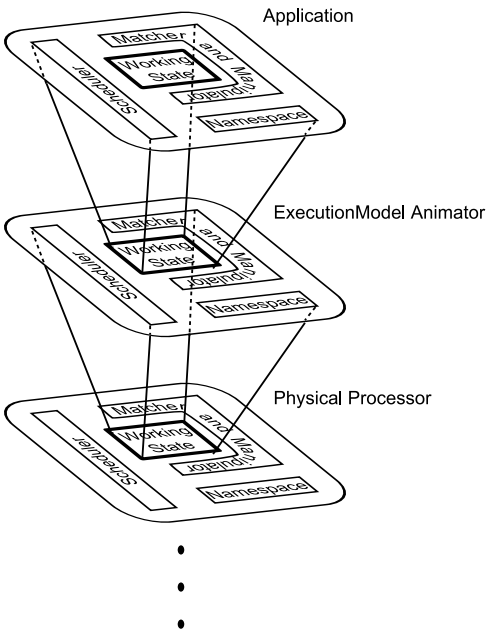

Figure 6.5: The ladder of animators, with the application as animatee at the top.

them. The relationship is that the animator loads its working-state with the internal-state of one of the processors in the tree. The processor, from the tree, currently being animated is the one whose internal state is in the animating processor's working state.

The animating processor remains hidden to all the processors in the tree.

6.2.4 More on the Nature of Time in the Model

In the statement "currently being animated", the "currently" means: it can be derived from the current state of the animator which animatee's working-state will change after enough changes of animator working-state. This brings up the question of the nature of time in this basic model of computation.

Definition of Time Inside a Processor: Given the rest of the model, time inside a given processor is purely local: a processor's time has no relationship to, nor effect on, nor connection

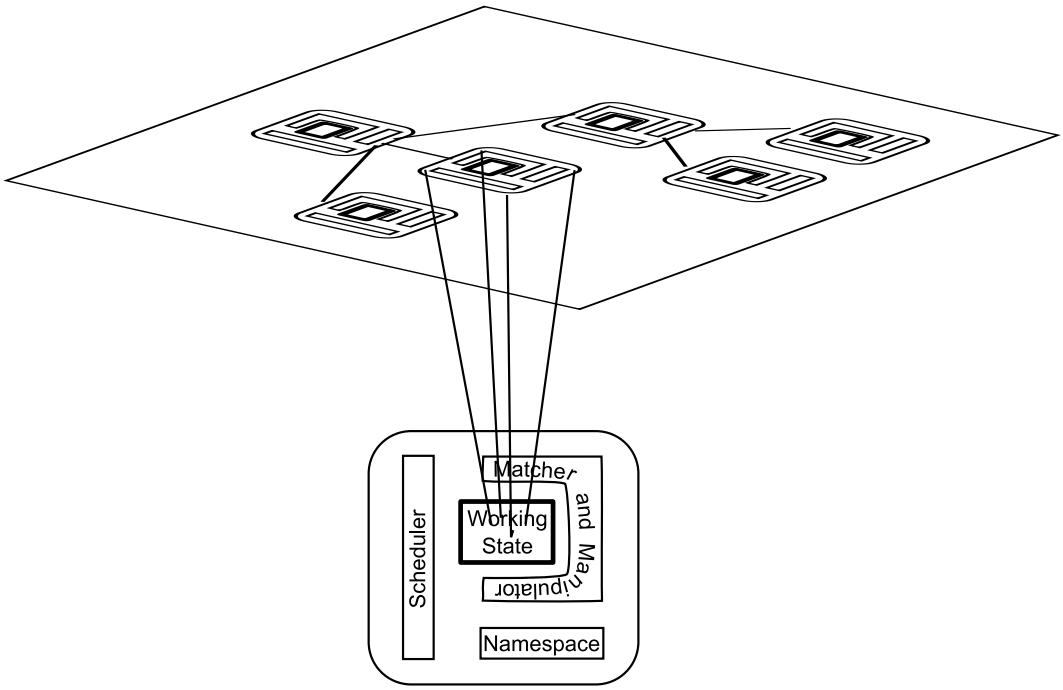

Figure 6.6: Animation vs Processor hierarchy. The tree in the plane represents the application-level processors created by function calls during a run. Each sees its children, which it communicates with via its namespace. They are animated one by one. The processor below the plane is the animator, which switches among them, animating each in turn. The application processors know the existence of their children, but are unaware of the animator, which is invisible to them.

to anything other than changes in the working-state. Therefore time is defined as the progression of working-state changes inside a processor. This implies that local time inside a processor is discrete. Each change of a processor's internal state is one step of its time.

Time Steps of a Processor vs Time Steps of its Animator: A processor has the state of its own computation, its working-state, but each of its elements also have hidden internal state. This implies a second time, which is hidden!

For example, one scheduling decision, with its accompanying removal of a pattern from working-state marks one change in working-state, which is one step of a processor's computation. But the scheduling decision required any number of state changes inside the scheduler. Each of

those was performed as the computation of an animating processor.

So two notions of processor state exist, and therefore two notions of time. But, looking back, it's clear that one is the time of the processor, the other is the time of an animating processor.

This will be an important concept when scheduling is studied in the context of specialization. There will be multiple levels of scheduler, corresponding to multiple enclosings of processors inside processors.

The VMS morphable hardware abstraction is an embodiment of this basic model of computation, and this difference between processor time and animator time plays a key role in the benefits of VMS.

In the next chapter, on the model of parallelism, a model for time among multiple processors will be developed.

6.2.5 Data vs Pattern

At this point, a precise distinction can be made between data vs pattern. A pattern is active, and a*part* of a processor. It relates to the actions of the processor in some way. In contrast, data is passive. It resides *in* a processor and is what gets moved through the namespace. However, data can be turned into a pattern, and a pattern can resolve into data, including data that is a representation of it, and so can be turned into a duplicate pattern! For data that is intended to represent a pattern, the only distinction between the data as data and the data as pattern is the context. The data has to be paired with active elements that provide behavior, and that combination is then a pattern.

For example, data that has passed through a namespace and been placed in a certain position in the recipient, can then be interpreted as a pattern by the recipient. No transformation of the data nor recipient processor-structure change has to take place.

This is a subtle point. According to this basic model, it is an implementation detail, because the model considers the combination of data with the active elements to be a pattern. The act of placing the data into a position where it interacts with active elements to produce behavior was the act of making that data part of the processor.

More precisely, a pattern defines boundaries: it is made up of a set of optional properties attached to boundaries, which define what is inside each of its inputs. Whatever is within an input's boundaries is defined as data. [16] An input can also bound nothing, which is an empty input, sometimes called a *hole*.

A pattern can be as simple as just providing boundaries, and nothing else. Or, a pattern can be more complex, for example, a rule-pattern has properties attached, which the scheduler takes as constraints that must be satisfied before scheduling the rule-pattern. It also has a property attached that states the type of processor to send the pattern's inputs to. For most rule-patterns, return data comes back, which is turned into a replacement pattern by the matcher-and-manipulator.

A good example of a rule-pattern is the `while` keyword in C. This pattern has, within one of its input-boundaries, the specification of the loop body. Within its other input-boundareis it has the result of the conditional-check. In the working-state, the loop body is treated as data. When the while-pattern is scheduled, the two inputs are sent to a "while processor". This is a processor that displays the behavior specified for the while keyword. It returns either empty, or the loop body data. Whatever comes back is turned into patterns by the matcher-and-manipulator.

Note that there is an implied semi-colon at the end of the while loop-body, which is another pattern. This pattern has scheduling constraints attached that prevents anything past the while-pattern from being scheduled until the while-pattern has been scheduled and returns

[16] Patterns are removed and what is within its input-boundaries is sent as data to a contained processor.

an empty.

Data has no effect. It is passive. To use data, such as to match it against other data, or use it as an integer or a floating-point number, it must first be turned into a pattern.

Note, though, that turning data into a pattern is different than receiving data into a pattern. The first creates a pattern, the second applies the boundaries of a pre-existing pattern to break the data into pieces.

Notice that a specification of a processor is data. It arrives at the creator arranged according to the creator's interface. The creator applies its boundaries to the data, breaking it into pieces. The creator then creates a processor containing patterns that correspond to the data values.[17]

These concepts of data vs pattern will be relevant in VMS and specialization. Specialization is about modifying specifications and mapping them onto multiple different physical animators. Data vs pattern is an important distinction there, especially when considering the point in physical time at which the scheduling choices are made for patterns implied by a specification. Some decisions are made before the program is run, either via static scheduling or via partial evaluation or symbolic interpretation. So, actions that are inherently part of a live processor are performed with the data that specifies how to make that processor!

Relating the model to physical circuits: A circuit element is a processor that has portions of its working state fixed, in the form of arrangements of atoms. The wires act as namespaces, while fields (voltages) moving down the wires are patterns in transit in the namespaces. The fields act as the inputs that get inserted into the working state, where insertion equals the field interacting with the atoms that represent the fixed portion of the working-state.

[17] This is getting into complexity that's not useful for this dissertation, but a particular value can be viewed as a particular pattern, and comparing values can be viewed as matching patterns. Hence, data is turned into a pattern before comparison is performed, making comparison an activity between patterns. Likewise a transform-type pattern like the plus operator can be seen as turning input data into patterns, then reducing those patterns to get the final returned value. Lambda Calculus has explored this fluid boundary between data and pattern.

6.3 Equivalent Circuit and Shape-Control Data vs Flow-Through Data

This section looks at Von Neumann processors as generating a circuit that is equivalent to the code in combination with certain parts of the data. It makes a distinction between data that controls the shape of that circuit vs data that flows through it. This is a valuable distinction in specialization because the shape-controlling data determines the amount of work performed by a processor.

6.3.1 Equivalent Circuit:

One way to look at the actions of a Von Neumann type processor is that it builds a circuit and passes data through it. An ALU instruction typically passes data through one portion of the circuit. A branch instruction typically chooses what shape of sub-circuit to build next. If it goes one way, the code there builds a circuit of a certain shape, if the branch goes the other direction, it lands at different code, which builds a different shape that gets added-on.

The context determines whether an instruction decides the shape of the circuit or whether it transforms data that flows through the circuit.

6.3.2 Shape-Control Data vs Flow-Through Data

The distinction between uses of an instruction gives rise to a distinction in types of data: Shape-Control data vs Flow-Through data. Shape-Control data affects the shape of the circuit, while Flow-Through Data flows through it.

For example, the condition input to a branch instruction is shape-control data, while the inputs to an ALU instruction are flow-through data.

Interestingly, both these types of data become patterns in the places where it interacts

with the instructions. The context causes the data to affect behavior, which makes the data, in combination with the instructions, be patterns.

This duality between data and patterns is the key to the simplicity and power of Von Neumann processors. They treat instructions as data in many places, then use the same data to control behavior in others. Places where it controls behavior data is a pattern, in combination with active elements in the context. The data values don't change, only the context. Taking that a step further, condition-code data can be viewed as becoming part of the stamped-out circuit pattern.

The key is that data alone is never a pattern, because patterns imply active capability. The data has to be in combination with active elements, and the combination is a pattern. This duality has some relevance to specialization, but not enough to look at it in greater depth.

The difference between between shape-control data and flow-through data is illustrated by search-intensive problems vs data-intensive problems.

Matrix-multiply is considered a data-intensive problem. The shape-control portion of the input data is the size parameters. They set the bounds on iterations in the loop nest. In addition, the current values of the iterators is shape-control data. The matrices themselves are memory-processors, each with a two-component namespace: a name is a row value in combination with a column value. The current iterator values determine the name handed to a matrix memory-processor, which gives back a flow-through value to be sent through the circuit.

6.3.3 The Amount of Work Done by a Processor

For matrix-multiply, just specifying bounds on iterations is enough to specify the size of the equivalent circuit, which is the same as specifying the amount of work to be performed. In general, the amount of work in a task is determined by the shape-control values. [18]

[18] Notice that a shape-control value is combined with instructions to yield a pattern. In general, the word "value" implies combining data with active patterns, and the value is the result of the combination. For example,

However, the values of shape-control data often correlate with the amount of flow-through data required by the task. For code where there is a correlation, the amount of work can be chosen by choosing the desired flow-through data – the required shape-control values are then derived from that. This is a useful concept in specialization, where dividing up work is a key activity. It allows dividing up the flow-through data in order to divide up the work.

To compare flow-intensive to search intensive, consider again matrix multiply. The equivalent circuit is highly regular. Its exact shape is determined exclusively by the shape-control parameters for matrix size.

In contrast, the circuit for Travelling Salesman is an irregular tree mapped out by the backtracking algorithm. [19] Each graph path is equivalent to one path generated in the equivalent circuit. The data that flows through the circuit-path is an accumulation of the edge-values from the graph-path. The amount of work is the circuit-size, which is determined by the graph shape.

Hence, the graph-shape is the shape-control data, and the particular shape determines the amount of work. For such NP-complete problems, the size of the graph, which is the number of nodes, has almost no relationship to the number of paths that have to be explored! So the amount of work done is unrelated to the amount of flow-through data.

So far, no accurate method exists to predict the amount of work from given shape-control and flow-through data, short of actually doing the work of constructing the equivalent circuit. Such a predictive method would, in fact, solve the P vs NP question, in favor of them being equal, so is unlikely to be found. This will be relevant to specialization of NP-complete problems.

32 bits can be combined with one set of instructions to act as four character values, or a different set of instructions to act a one integer value.

[19] or whatever heuristic is being used, which always contains embedded backtracking, except for some that settle for a guess to the answer, rather than insisting on the exact answer.

6.3.4 Grouping Code by its Use of Data

The preceding suggests ways of grouping programs:

- The degree of regularity of the equivalent circuit
- The amount of input data required to fully control the shape
- The amount of work required to predict that shape, given the data
- The relative amount of work done to decide the shape vs flow data
- The point at which the shape can be determined (static vs init vs input present vs fully complete)

These characteristics affect specialization. For example, in the case of Travelling Salesman, new shape-control state is generated by a control-instruction interaction of previous state with the input shape-control data. The fact of being a control-instruction that generates the shape-state update means that the shape is dependent upon non-simple details of the input data, and in general cannot be predicted in a simple way. [20] Specialization will use this conclusion to make choices important to the parallel performance.

In contrast, the shape-control state in the matrix-multiply has a simple and predictable update, and also determines the name given to the memory-processor holding the flow-through data. This fact that the shape-control data does both indicates correlation between flow-through data consumed and particular values of shape-control inputs. This is important to specialization choices.

The concept is important, that shape-control state interacts with the code to generate a circuit that flow-through data flows through. It will be referred to in the Holistic Model,

[20] The fact of being a loop of new shape-control generated from old shape-contol, is key, and when combined being updated via control-instruction, that involves input data – that pattern is a good sign that predicting circuit shape will not be practical, which implies that predicting the amount of work performed will not be possible.

specialization, the BLISS framework, and the design of programming languages.

6.4 Execution Models

A programming language is a collection of many things. One of them is an execution model, which essentially states "how execution is performed". In terms of this basic model, an execution model can be defined more precisely as a statement of two things: 1) a set of primitive processors, including the interface of each, the name of its specification, and a description of its observed behavior. 2) The execution model also includes the interface of a creator. [21] The creator interface implies the format of processor specifications (for example, the syntax and grammar of function definitions), which in turn implies the behavior of the three elements of processors created.

The execution model itself only states interfaces and observed behavior. The implementation determines the details of how the three processor elements interact, the format of the working-state, the format of patterns in it, and so forth. These are all set by the implementation of the creator, which creates those elements inside execution-model-processors (ExeMProcessors). Normally, an execution-model-implementation (ExeMImp) is stated in terms of the interface of a target animator, which will animate the created ExeMProcessors.

A compiler includes the implementation of the language's execution model. For most languages, it performs re-stating the processor-specifications in terms of the target animator. For nearly all languages, their execution model includes behaviors that a typical physical processor does not directly implement. The language implementation includes a so-called runtime, which implements the missing behaviors of the target animator. [22]

The runtime implements the behaviors of the three elements that are not directly

[21] The creator is part of the namespace of the processors created by it – which is self-recursive! The top-level processor is created by the OS, by using a creator provided to it by the implementation of the execution model.
[22] Even C includes a runtime.

implemented by the physical processor, which will ultimately animate the three elements. The interesting thing is that the runtime is itself animated by the physical animator, which makes for an odd animation-hierarchy. The physical animator has to switch from animating the portions of the ExeMProcessor it can directly animate, to animating the runtime, which in turn animates the other portions of the ExeMProcessor.

An implementation of an execution model also includes a set of primitive processor implementations. The model itself only states the observed behavior of these processors, as seen from the interface. It is the implementation of the model that has the processor specifications, normally already in terms of the target animator. The processors created from the specification, and animated by the target animator then exhibit the model-stated behavior.

As an example, a dynamic language like Perl has an execution model that allows modifying the specifications of processors after the top-level processor of the program has been created (in other words, during the run of the program). [23]

6.4.1 Applying the basic model to Von Neumann processors

The machine-instruction-values of a Von Neumann program are considered data at the point they are given to the OS to create the top-level ExeMProcessor. Because of the nature of Von Neumann physical processors, all the code becomes patterns at the point the top-level ExeMProcessor is ready for animation. [24] However, each function-call creates a new ExeMProcessor, creating the variable parts of its working-state on the call-stack, and inserts the inputs (parameters) into the working-state by placing them onto the stack in a certain pattern

[23] The point at which the top level processor in a program is created is considered the point at which the program begins a run. For a language like C, the OS directly creates it. However, for dynamic languages, the creator provided to the OS first creates an animator for the language, as the OS-created processor, and then has that animator animate creating the top-level processor. For interactive programming the top-level processor interacts with the programmer, for runs, it is the top-level processor of the program.

[24] unless the program performs self-modification

(the calling convention followed by compiler). [25]

So, the code inserted by the compiler that makes room on the stack is an implementation of part of the creator of the calling ExeMProcessor's namespace. The code that places the parameter values on the stack is an implementation of the calling processor's namespace, which has transported the values to the new ExeMProcessor, and the particular pattern of parameters on the stack is in accordance with the new processor's interface (specified number of parameters in the function-definition). It is part of the details of ExeMProcessors determined by the creator, and chosen by this particular compiler's implementation of the language's execution-model.

Notice that many of the instructions executed just prior to the `call` instruction are not part of the patterns of the calling processor! Instead, they are preparing space on the stack and inserting parameters for the called ExeMProcessor. So, these instructions are actually part of the animator of the calling ExeMProcessor! They are part of the patterns in the processor animating the calling ExeMProcessor. What has taken place is that the physical processor went from directly animating the calling ExeMProcessor to animating the execution-model's runtime processor. The runtime processor's patterns encode the namespace implementation of the execution-model.

What happens next is interesting: the call instruction causes the physical processor to stop animating the runtime processor that is animating the calling ExeMProcessor, and switch over to animating the runtime processor animating the called ExeMProcessor! This new runtime processor creates the space on the stack for the variable parts of the working-state of the called processor.

A C program has an implied processor that it begins with, which is constructed ac-

[25] This description of creating a new ExeMProcessor and switching to it is simplified, for explanation purposes. The model still applies to the more complex way that modern compilers implement the runtime portions of the code, but explaining that is more confusing and doesn't move towards this dissertation's goal of using the model as a thought-aid.

cording to the main() function specification [26]. Each function defined in a program is the specification of a processor, and each function-call causes the creation of a processor of that type, followed by communication of the parameters to it. Local variables exist in the processor's working-state (on the stack), and any non-local variables are memory-processors reachable from the processor's namespace[27]. During the call, the function-processor communicates with the memory-processors that are the non-local variables. When the call is complete, the created function-processor dissipates. The results were either communicated back via a return statement, or reside in the memory-processors communicated with.

The call-stack holds the state of processors that have been created. Each has its animation suspended as part of the function-call invocation. Only the last processor on the stack is animated [28].

So, a sequential program constrains scheduling such that only a single processor is animated at a time, which allows a single lower-level processor to animate those in sequence. That is the essential feature of a sequential program. It allows the entire program to be animated by a single physical processor. And that single, physical, animating-processor has only a single time-line.

6.4.2 Imprecision is Intentional

As seen during these various applications, the basic model is imprecise, but the context it is applied-to fills in details. This is intentional and useful. The insight provided so far and

[26] The OS is itself a processor that performs the creation of the top-level C processor that corresponds to the "main" function. This main processor exists in the OS processor's namespace.

[27] As an example, static variables are created, during compilation, in the data-segment. Allocating this space corresponds to creating a memory-processor with the static-variable's name – or, alternatively, corresponds to adding an entry to the pattern namespace of the global-memory-processor.

[28] unless an interrupt or fault takes place, in which case this last one is suspended as well, and a processor from the OS stack is animated. Each stack is associated with a single lower-level processor that is in turn animating the processors in the stack. That lower-level processor's state is held in the physical processor registers, and is what the interrupt mechanism modifies. Changing this processor-register state is what switches animation between the stacks.

in the next two chapters is made possible by this delay of precision. It allows the same basic patterns to be mapped onto very different details. The value lies in having those common patterns across such different contexts.

Insistence on ultra-precise models, such as one finds in operational semantics, or mathematics itself, would take away this ability to find commonality, and so reduce the usefulness. In fact, most useful things in the real world happen without high precision. The notion of a precise model is a human invention, which has been very helpful, but does not exist outside of agreement among people. Such models allow proofs and the applications that mathematics and computer modelling have been put to. However, the usefulness has largely come from indicating how to simplify the environment, by controlling it such that the equations and models used in engineering apply to the controlled context.

In contrast, the model of basic computation given here has more value by not being precise, by delaying that until application. It's use is a dual to that of most models in engineering, which state how to control the context to conform to the model. Here, the model is controlled by the context, which makes the model conform to it! The model's four essential features are controlled by the context, which fills in the detailed definition of each.

The context in which the model is being applied determines the exact definition of the scheduler behavior, the format of a scheduling constraint, the mechanism that makes up the namespace, and so on. The commonality, which provides the usefulness, is that there is a thing that translates name and communicates to the named, a thing that holds patterns, a thing that checks constraints and schedules patterns by handing off to the name-thing, and a thing that does insertion and removal of patterns. It is possible to precisely define those things in many different ways! Take details from the context, then see what those essential things map onto.

Self-Recursive model: An interesting observation about the model is that each of a processor's three active elements can be recursively modelled in terms of a processor. In fact, this is what animation is: the animator is a processor that exhibits the behavior of the active elements of a different processor. So, for example, the scheduler can be stated in terms of a processor that itself has the same three elements inside it. Any of the three elements can be stated this way.

This self-recursion of the model is satisfying because the most primitive model of computation should not rely on the magical existence of even more primitive elements. The bottom level should be describable in terms of itself.

This self-recursion also allows time to be defined only as a sequence of changes in the working-state. Nothing more. The working-state changes occur due to the three active elements, and their activity is due to changes in working-state of an animating processor.

6.4.3 Scheduling is the Difference In Sequential vs Parallel Programming

The difference between sequential programming and parallel programming is the scheduling process. Sequential programming always includes enough extra scheduling constraints such that exactly one task is free to be scheduled at any point in time. While parallel programming eliminates many of the unneeded [29] constraints, allowing multiple tasks to be scheduled to overlap at points in time.

However, time is no longer a single entity in parallel programming, as it was for sequential programming. In sequential programming, there is exactly one time-line defined for the program, and each statement is one step in that time-line [30]. However, for parallel programming,

[29] unneeded means that the correct result is still obtained without them, although they may be needed for non-correctness goals

[30] This is ignoring I/O which actually introduces a simple form of parallel programming

time is no longer singly defined, there exist multiple time-lines.

Relating this to the basic model of computation, a sequential program only has a single processor free to be animated, at all points in any timeline. This allows mapping all of the processor timelines onto a single timeline, despite having multiple independent processors in existence.[31] This property makes animating a sequential program with a single physical animator very simple.

When going to parallelism, multiple physical animating processors exist [32]. Each of those animates a sequence of processors, within its own time-line. So, multiple animator time-lines exist.

This in and of itself is no problem. However, the function-call processors may each communicate with the same memory-processor. The order of these communications can affect the computation result!

This means programming constructs are needed to control the order that the various processors are animated, in order to control communication with a given memory-processor. The communication order must be constrained to give the desired computation result.

These new constructs are the programmer-visible difference between sequential programming and parallel programming. In this dissertation, such constructs are called *parallelism constructs*.

The Thread constructs mutex, lock, semaphore, condition variable, and so forth were the first such parallelism constructs. Later ones include synchronous send-receive, which corresponds to pi-calculus operations, spawn and sync, publish-subscribe, and so on.

This brings full circle back to scheduling and time lines. In sequential programming,

[31] The stack holds the variable portions of the working-state of these processors.

[32] such animating processors can be one-to-one with physical processors, or they can each be a different stack that a physical processor switches to. A Thread is implemented as such a stack, and is effectively defined by its implementation. Thread implementations use some form of hardware support to switch the physical processor between animating the different stacks.

multiple processors may exist on the stack, but the scheduling constraints allow a simple mapping of all their timelines onto a single timeline.

In Parallel programs, wording becomes clumsy. There is no longer a single time. This eliminates the ability to have a global view of what's happening in the program as it runs.

One way to get a global timeline is to define static variables and the heap as being in a single memory processor, and using that processor's timeline as the global one. However, this has limitations, most notable, it's only valid for one specific kind of physical processor architecture. It's especially clumsy when considering the existence of multiple caches and the need to talk about data moving between them.

A second way to get a global timeline is to invent a virtual timeline that has limited properties. This was the approach taken with VMS, as seen in Chapter 11.

Chapter 7

Holistic Model of Parallel Computation

This chapter applies the basic model of computation to parallel hardware and the portions of system software and application code that act as animators of execution models. The point of the chapter is to expose structure within computation on parallel hardware, to inform specialization.

It starts by describing parallelism. Then develops a model of computation on parallel hardware. It goes on to use the model to perform thought experiments, which expose what abilities of schedulers affect performance, and the characteristics of applications that feed those scheduler abilities. The next chapter is then informed by those application-code characteristics and the scheduler abilities they enable, and talks about the process of mapping a single parallel source-code onto multiple different parallel hardware targets.

The model is called The Holistic Model of Parallel Computation, because it takes a system-level "holistic" view. It centers around scheduling, and focuses on the interactions between various scheduling approaches and particular code characteristics, which are in turn modulated by hardware parameters.

7.1 Overview of the Model

The model introduces a simplified loop, by which parallel computation progresses. Scheduling is at the heart of the loop, determining what the tasks are, what processor animates each, and at what times. The scheduling decisions cause work, including its data, to be sent to the chosen processing elements, and when the work is done, status is sent back to the scheduler. The status update triggers new scheduling decisions, and repeat.

The quality of the scheduler decision determines the network behavior delivered to each scheduled task. This communication time plus the scheduler's think-time and the work time set the round-trip time through the loop. However, man loop iterations overlap, one iteration for each task. It is the pattern of overlap that determines the total running time.

7.1.1 Holistic Scheduler is in the Animator

The point of the model is to bridge the gap between code and the hardware it runs on. As seen in the previous chapter, starting at the bottom, hardware animates the execution model, which in turn animates the ExeMProcessors of the application. As such, the scheduler featured in the Holistic Model is the scheduler in the animator, not in the ExeMProcessors of the application.[1]

7.1.2 What the Model Shows

Thought experiments performed with the loop show that certain abilities in the animator's scheduler enable higher quality decisions. The consequences of these scheduler abilities

[1] One complication is that many applications include code segments that are actually part of an animator. This happens when the language doesn't have semantics for parallelism, such as C, or when its semantics allow application programmers to directly modify the animator, such as Sequoia. Parallel programming is performed in C by escaping the C execution model, via library call that invokes hardware, which is how pthreads works. These libraries usually are so primitive that the application implements its own level of animator above them. Having animator code in the application also hurts portability.

are explored, to predict the effect given combinations will have on computation time, hardware utilization, or energy. Abilities explored include: ability to divide large tasks[2] into smaller ones or group small into larger, keep track of where data resides, predict what data a task will use and produce, and how long it will take to execute.

Each of those abilities affects quality of scheduling, which impacts the goal measure: better decisions mean better performance, or lower energy, and so on. The nature of the application and hardware determines what abilities are enabled, then the scheduler implementation determines which of those it takes advantage of, plus how much time it spends making the decisions. This triple interaction among enabling decision qualities vs taking advantage of them vs cost of making decisions is what the model is all about, and provides many of the insights needed for specialization.

The rest of the insights needed for specialization come from considering how the hardware influences the interactions. It essentially moves the cutoff points at which taking advantage of particular abilities becomes too costly in the scheduler and is no longer a net win. Scheduling decision time should remain much shorter than communication and computation time in order for the scheduler to make parallel computation profitable. [3] The work-units made available by the application are tested against the cutoff points to decide if they should be scheduled in parallel vs sequentially.

[2] Task was defined in the previous chapter to be an amount of work equal to an equivalent circuit. The circuit, and therefore task, is defined by a segment of code plus shape-control values and the data that flows through. It is also equivalent to a single consumable ExeMProcessor, possibly for an internal execution model defined by the specializer (such an internal model allows discrete tasks to be defined for circuit-type languages, and also allows resizing tasks).

[3] As seen in the basic model, multiple levels of processor can exist in hardware structure. Processors in each level have schedulers, implemented either in hardware or system-code. As seen in the next chapter, effective specialization will insert processor levels into the execution model's animator, to match levels in hardware. Profitability of parallel execution is considered at each of those levels.

7.1.3 How Hardware Affects Performance

The pattern of communication paths in the hardware impacts cutoff points the most. This is because the hardware pattern interacts with the communication-request pattern. The interaction determines the time and width of communication flow. The communication-request pattern is, in turn, the result of the scheduler choices, meaning that the hardware the scheduler cares most about is the communication paths.

Hence, communication paths in hardware set the framework the scheduler operates within. The primary determiner of performance is how the scheduled communication patterns interact with the hardware communication paths (and hardware network schedulers). So, the primary concern in the schedule-choosing process is the communication pattern implied and how that will interact with hardware communication paths.

It will be seen that this causes specialization to revolve around scheduling: identifying the processors in software and in hardware, predicting their interactions, then iteratively modifying the software. Each iteration chooses schedulers for each processor level, and predicts interactions again. This repeats until run out of time, or no improvement can be gained.[4]

7.1.4 Simplifications of the Model

The Holistic Model has to simplify, in order to show cross-hardware trends. So it doesn't consider the details of the particular scheduling choices, nor their interactions with hardware paths. Instead, it uses an abstract quantity that represents the quality of that interaction. The quality represents the outcome from the details, abstracting away the particular interactions by which that outcome emerges. As a result, the achieved communication is a function of the schedule quality-level.

Thought experiments are done to illustrate the result of trading communication achieved

[4]Such iterations should be familiar to compiler writers, as many optimization approaches use them.

vs quality-level of scheduling at various think-times. This is repeated for combinations of scheduler abilities, enabled by the application, with various quality-vs-think-time relations for the scheduler.

Tying the Model to Real Hardware and Software: The chapter rounds out by considering typical hardware and software, in which multiple levels of communication exist, a scheduler for each. Some of the scheduling is hard-coded, such as inside cache-controllers, and other scheduling has traditionally been included in the application, such as explicit message passing and the use of lock-based synchronization primitives. The consequences of these features is considered in light of the Holistic Model thought experiments.

7.1.5 What Want from the Model

The overall purpose of this dissertation is to give a recipe for meeting the three goals for parallel software. To find such a recipe, the structure of parallelism needs to be known, and then the interactions among the elements of that structure. It is a hierarchical structure, with the categories of parallelism as the top level elements: parallel hardware, parallel system software, parallel languages, and parallel application code, then sub-structure within each of those.

The Holistic Model was developed to reveal the structure of computation in parallel hardware, and from that identify interactions with the other aspects of parallelism. Although it is a model of computation on parallel hardware, to be complete, it includes the portions of software that perform scheduling of work onto processing elements.

The basic model of computation is applied to parallel hardware. To get a single model that covers (nearly) all parallel hardware, simplifying assumptions are made. Because the purpose is to gain insight, rather than exact numeric predictions, the simplifications only have to

preserve the implications for specialization. Because only thought experiments are being performed, it is personal judgement of each reader whether they feel the simplifications preserve valid conclusions.

7.2 What is Parallelism?

In order to model something, a good description should be had of what is being modelled. In this section, various aspects of parallelism are briefly described, with focus on what the Holistic Model will include and be useful for.

7.2.1 Hardware View of Parallelism

Parallel hardware is diverse, much more so than sequential hardware. It tends to contain many levels of communication, each of which defines a namespace and therefore a (basic model) processor that also has a scheduler. Such schedulers are normally part of the hardware, with a hard-coded scheduling algorithm.

For example, a cache is a separate processor by itself,[5] and the scheduling algorithm is the combination of replacement policy and associativity. This scheduler generates communications when a requested address is not found inside.

The memory interface of a GPU processor makes another good illustration. It defines the interface to a memory type of processor. In GPUs, this interface is highly complex, requiring very wide requests that are given back in chunks over time. This is because the GPU's processors are SIMD, so a single memory instruction must supply an operand for all of the SIMD processors, in a single chunk, and the SIMD processors are also pipelined, so a single memory instruction has to supply several such chunks in sequence.

[5] To be precise, a cache has normally an internal hierarchy of processors: each block is a separate namespace, and the blocks are themselves within a namespace, and so on.

Thus, the memory processor has a hierarchy of processors. The top-level has patterns that cause multiple schedulings to the DRAM memory-processors. and also multiple schedulings back to the requesting SIMD processor, one for each returned chunk.

Another example of parallel hardware is the traditional super computer, with some communication topology connecting many processor cores. A simple example would be a mesh connecting nodes. Each node contains multiple cores on a bus, sharing a main-memory, and each has its own L1 cache.

Each node in the mesh acts as a separate namespace, and these are connected into a larger namespace by a routing algorithm. Hence, this acts as two levels of processor, one for the larger namespace, which has patterns that perform the routing, and a second level whose interface is a north, south, east, or west request, and has patterns that turn that into signals on wires.

In addition, inside a node, the bus acts as a namespace, defining a third level of processor. The patterns here are the bus protocol. The bus delivers to the main-memory processor on one end and the caches on the other, so both main memory and caches are a fourth level of processor.

Finally, the individual cores have a register set, which defines another namespace, adding another level of processor, and then the internal datapath.

This many levels makes for complex interactions between levels of scheduler, whose scheduling decisions cause communications within the various namespaces (networks). The scheduling decisions made in the system software have a very complex set of communication-requests they cause to happen across the various levels. This makes it difficult for the scheduler to achieve high quality decisions!

7.2.2 Programmer View of parallelism

The programmer sees an execution model that abstracts away many of the levels of processor in the hardware.

Some execution models require the programmer to implement at least part of the animator of the application processors. In essence, the application adds a level of execution model on top of the one provided by the language[6] or library calls. An example is Sequoia [28].

Such languages give the programmer no means to state the scheduling constraints explicitly. Instead, the programmer encodes enforcement of them in their animator implementation.

A variation on this, which often appears in C programs using pthreads or MPI [62] calls, is mixing application-processors with their animators. Such a Multi-threaded C program defines the equivalent of an application-processor, including its namespace implementation and scheduler implementation, by placing control constructs that enforce scheduling constraints on lower-level application processors.

However, these control constructs surround pthread calls that suspend and resume and create communication between the threads animating the scheduled application processors. These calls make this portion of the application code be part of the animator of the application processors. As a result of this mixing, pthread code is exceptionally difficult to think about and get correct. It is equally difficult for toolchains to reverse-engineer to extract the scheduling constraints.

Other execution models, such as CnC [24], and DKU (described in the chapter on BLISS), expose only explicit scheduling constraints to the programmer. The toolchain is free to rearrange the structure of processor definitions and creations as convenient, as long as the

[6] Libraries that provide a separate execution model, such as pthreads and OS calls, are considered an embedded language, following the definition of language in Chapter 6.

programmer-stated constraints are satisfied.

7.2.3 Toolchain View of Parallelism

The toolchain sees the source code, forms internal representations of it, modifies that, and generates new code with a different structure. The new code respects all scheduling constraints, [7] but the structure of the processors defined by the new code may be quite different.

In addition, a target animator for the language is inserted, including the scheduler inside that animating-processor. For hardware with multiple levels of communication hierarchy, the target animator for the language is likely to also have multiple internal levels of processor, to match a scheduler to each level of the hierarchy.[8]

The toolchain will need to derive quantities and curves, which it then uses to predict the best transforms[9]. This chapter in combination with the next will show that these end up being equivalent to the parameters of the Holistic Model, making a the model a good basis of how the toolchain sees parallelism.

7.3 The Holistic Model of Parallel Computation

The basic model is given, here, first, then in the following section its behavior is explored with thought experiments. In the section after that it is connected to actual hardware and applications.

[7] Except constraints that can be proven to be superfluous, which is what parallelising compilers for sequential languages must do. Such a proof is practical only for a few special cases, and in general appears to not be possible in practical time-budgets.

[8] For the lowest levels of hardware, such as the register set level, the scheduler decisions are made statically, in the toolchain. For higher levels in the animator/hardware, code is inserted that acts as the scheduler during the run. Recall that the animator is hidden from application processors, so the toolchain is free to do insert such dynamic animator code without modifying application code.

[9] toolchain transforms are equivalent to changing static scheduling choices implied by sequential code – which modifies the communication-request pattern – and also to changing the processor specifications, which modifies the work caused by a given scheduling choice.

Single Level of Scheduler: As illustrated in the previous chapter, a hierarchy, of processors, exists for most processing. Code running on parallel hardware is no exception. Multiple levels of hierarchy exist not only in the the application code, but also in the hardware. Each processor in each level has its own scheduler.

The application processors are animated. The animators are mapped onto the hardware. Because this model's purpose is to gain insight, it simplifies. It treats application-processors simply as tasks, which are scheduled by the animating processors. It abstracts away the details of tasks, and only considers behavior of the animating processors in detail. In addition, it only considers the effects within a single level of the animator hierarchy at a time.[10]

7.3.1 Inputs to the Model

Base Quantity is Quality of a Scheduling Decision: The model abstracts away the detailed interactions between scheduling choices and the hardware. It represents the outcome of task-hardware interactions as a quality value, the quality of a scheduling decision. A schedule has multiple types of quality. This section considers only two: communication quality, and task-choice quality.

Quality vs Think-Time Curve: The model takes as input a curve for schedule communication quality as a function of think-time. In general, this curve is different for each scheduling decision, and depends on all previous decisions, as well as all future ones. Various levels of simplification can be applied, such as assuming an average curve for the whole program run. What matters is that the simplification preserves behaviors that matter to specialization choices.

The quality curve for a given context also depends on the nature of the scheduler implementation. As will be seen, the quality of schedule a scheduler can produce is capped by

[10]This is too strong a simplification for many hardware-application combinations of interest. Future work will perform experiments that illuminate the implications for specialization, of interactions between levels of animator hierarchy.

its ability to predict things such as resulting communication requests, amount of work in a task, and so on. Each combination of abilities in the scheduler enables a different quality curve.

Communication vs Quality Curve: A chosen schedule's communication quality is plugged in to a latency-and-BW vs quality curve, which is another input to the model. The particular task scheduled sets the amount of data communicated. The latency and bandwidth applied to this then determine how long until the task fully arrives and is ready to run.

Work per Time: Computation on a processing element is modelled as a work-per-time value. In general, this value depends on the past scheduling decisions, and for some hardware may also depend on future ones[11]. The choices of schedulers at other levels in the animator/hardware hierarchy also affect this value, in general. [12]

7.3.2 Overview of the Model

Overview: The model centers around scheduling. It treats computation as progressing in a loop that takes place around the scheduler: the scheduler makes a decision, which causes communication, and then computation, and then communication of status back to the scheduler, which makes another decision, repeating.

Scheduling decision: In more detail, the scheduler chooses work, a processor (computational element) to perform the work, and at which point it will issue the decision. At the chosen point, the work starts being communicated to the processor, which performs the work, then notifies the scheduler of completion.

[11] example, out-of-order pipeline where a load instruction may hit in cache because of logically future instructions that happened to jump ahead

[12] Making the computation-per-time value invariant may be too simplistic to give good implications for specialization on machines with many levels of hierarchy. Future work will perform experiments to expand the model to include interactions between levels of the hardware hierarchy, in order for specialization implications of the model to hold.

The communication quality of the scheduling decision determines the network behavior, which combines with task data-size to set the communication time. The total loop time is that communication time plus the time the scheduler spent thinking (plus work time and status comm time).

Each task scheduled is enclosed in such a loop. So, multiple iterations of the loop overlap. It is the pattern of overlap that determines the total running time.

The best possible decision in a given context requires looking at all possible future consequences of the decision, so it's not a practical quantity to measure. However, the idea that some scheduling choices are better than others, is a valid concept. The complication is that the context is the result of past scheduling decisions. Hence, the best is only the best if all the preceding and all the following decisions are also the best.[13]

Scheduler Think-Time: The scheduler takes time to come to a decision. In the ideal case, the longer it thinks, the higher quality its decision, but unfortunately, the non-linearities involved in real hardware make this not true. In addition, some code has a smoother quality-vs-think time than others.

Communication: The model treats communication as a relation between scheduling decisions and the resulting latency and bandwidth of communications. In general, the relation should take as input a state resulting from past scheduling decisions, plus a current scheduling decision, then yield a latency and bandwidth of each communication generated by the schedule (some placements cause more communication initiations than others).

In the hardware a communication's time depends on past scheduling decisions, as well as the particular end-points of the communication. The physical topology determines the

[13]The best in a given context may actually give a worse overall result when none of the other decisions are the best. This is a fundamental phenomena.

resources that can carry data, and the scheduling built into the network determines how those are used to get the data between the end-points. Past scheduling decisions cause some of the resources to already be in use. So past scheduling in combination with the network's own scheduler determine which resources are available for the current communication.

The model abstracts these details away. The strongest simplification is to just consider a single average latency and bandwidth, regardless of scheduling history. This actually works reasonably well for a significant number of network-application combinations.

Another simplification is to consider the point-to-point average over some benchmark set of scheduling decisions. This also works reasonably well for some hardware and some applications.

What matters is that the model preserve the behavior that informs specialization activities. The purpose is to learn what inputs specialization needs, how to take that input from the code-base, and how it should use that input to affect the code. The communication model doesn't need accuracy, it only needs to preserve emergent behaviors that inform these things.

Generating a communication vs quality curve: In the simplification used in this section, each communication curve represents a different combination of application with hardware. The detailed interactions determine the curve shape. Each point on a curve is the bandwidth and latency delivered to one task, for one particular schedule for that task. The curve is generated by all possible schedules being sorted by the communication they result in. The resulting ranking sets the schedule's quality value.

Thought experiments: This dissertation only uses representative curves that illustrate a particular feature that a real one might have. These are used in thought experiments to gain a feel for implications on specialization.

A later section will subjectively examine various kinds of real hardware. It will predict the shape of communication curve for the hardware in combination with some kind of application characteristic values (like intensity of requests). This will tie various kinds of real hardware to the model and then the relevant thought experiments will suggest what specialization should do for each kind of hardware.

7.3.3 Interactions Among Elements in the Scheduling Loop

The scheduling loop is illustrated in Figure 7.1.

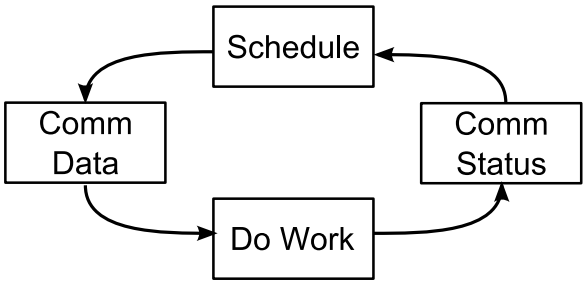

Figure 7.1: The scheduling loop of the Holistic Model.

Action begins in the "schedule" box, where the scheduler picks a task (an application processor to animate, defined by code plus shape-control values and all consumed data) and assigns it to a physical animating processor (this can be a hierarchical processor with others inside its namespace).

The schedule is released, by the scheduler, which invokes the second box. Here, the communication network (the namespace of the processor doing the scheduling) performs transfer of all data to the processing element chosen to animate the task.

The communication box is symbolic, or logical. It represents the total time taken to perform the communication. This doesn't all necessarily happen at once. For example, when the task is assigned to a core plus cache, the communication will be spread out, by the cache,

over the course of the entire task. This is all collected into the Comm Data box.

Logically, following communication is the computation, which happens inside the Do Work box. All the computation time of the task is symbolically collected in this box.

When the task is complete, the processing element sends notification back to the scheduler. The network time for this is inside the last box.

One iteration of this loop exists for each task performed in the application run. These iterations overlap, in the scheduler's timeline. An iteration begins with the scheduler being triggered to make a scheduling decision. The total time is the sum of the scheduler think-time, the data comm time, the work time, and the status comm time. It is the pattern of overlap of these iterations that determines the total time for the run. The more overlap, the shorter the total running time.

7.4 Thought Experiments

This section chooses some curves for communication vs quality of scheduling, and for quality of scheduling vs think-time. It plugs those in to the loop, and looks at the consequences.

This section introduces complexity slowly, building it up factor by factor. First it just gives communication vs schedule-quality curves without considering any other factors. Then it shows how a particular schedule chosen has a particular quality, which picks one point on such a curve. Next it adds the fact that the scheduler spends time picking the schedule, so quality is a function of time spent thinking. Then it combines the two, to yield communication vs think-time. Next, the scheduled tasks's data-size has to be plugged in to the communication delivered to the task, to yield total communication time. That time is added to work-time, plus think-time, plus status feedback time, to yield total time of one iteration of the scheduling loop. Finally, to get total running time, the overlap of the iterations of the loop is factored in.

7.4.1 Thought Experiments on Effect of Scheduler Abilities

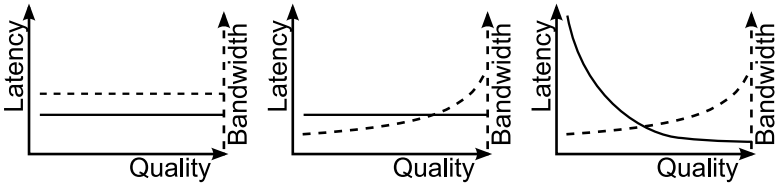

Figure 7.2: Three curves for latency and bandwidth as a function of quality of chosen schedule. Each curve represents a different combination of application with hardware. The detailed interactions of the two determine the curve shape. Each point on a curve is the bandwidth and latency delivered to one task, as a result of the detailed interactions caused by one particular schedule for that task. The curve is generated by all possible schedules being sorted by the communication they result in.

Figure 7.2 shows the communication vs quality curve for several hardware-application combinations. What the curves show is that the effect of quality on communication can range from none to extreme.

The first curve is flat for both latency and bandwidth. This would happen for a hardware network that is fully connected, or hub-and-spoke topology. In reality it would be more complicated because some hardware paths may already be in use, which introduces a step in the curve: a low-quality choice waits for the link to be free, a high-quality choice picks a different node that has all links free.

The other two curves show that quality of scheduling choices can have a very strong impact on the network latency and bandwidth the scheduled task receives while moving its data.

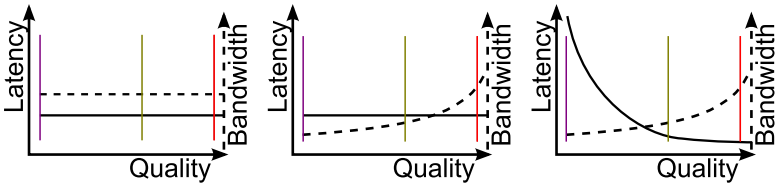

Figure 7.3: Three schedules are mapped onto the latency and bandwidth curves. Each schedule has a quality that picks out the communication delivered to the so-scheduled task. Think-time used to arrive at a chosen schedule is not considered here.

Figure 7.3 shows what communication is the result, for various scheduler implementations. Each vertical line represents the quality of the choice made by one scheduler. The heights at which it intersects the latency and bandwidth curves give the latency and bandwidth delivered to the task so scheduled.

The schedule, for this simplified illustration, consists only of which processing element the task is assigned to. The data in the task may be on a different processing element, or in main memory, or in an intermediate memory processor [14]. The latency represents the time from activation of the (decided-upon) schedule until the first data of the task arrives at the assigned processing element. The bandwidth is the amount of data per time given to the task during its data transfer.

In real hardware and software, such curves would be more complex, and they would change for each task scheduled, based on the communication resources already in use

The first scheduler, purple line on the left, is the simplest possible: it takes the first available task and assigns it to the first available animator. It doesn't do any search nor take into consideration the location(s) of task-consumed data.

The second scheduler adds the ability to predict the locations of inputs, and predict communication times. It has a simple algorithm for tracking and predicting, but the quality has improved.

The third scheduler is perfect. It adds the ability to predict communication time, execution time, and future scheduling decisions. It produces the best possible scheduling decisions. The difference in communication is evident.

Because these curves are made-up, the relative spacing of quality of the schedulers has no meaning, and the exact shape of the curves has no meaning. So this can't be used for accurate prediction of resulting communication. However, their general shape is informed by experience

[14] a later section will look at estimating the curve for given real hardware running a given real program

with actual hardware and actual programs. The trends of the effect of scheduler abilities upon communication correlate with observation.

Conclusions: The conclusion that predictive abilities used by schedulers improves communication delivered to a scheduled task correlates with observation. Also, the conclusion is valid that communication's sensitivity to scheduling quality is determined by the nature of the communication hardware in combination with the application. The combination sets the curve, from which the quality picks values.

7.4.2 Combining Communication Curves with Quality Curves

A scheduler with predictive abilities performs a search for the best choice, so it spends time performing the search. Normally, it can stop that search at any point. The longer it searches the better the decision it generally comes to. This relationship is represented as a graph of quality vs think-time.

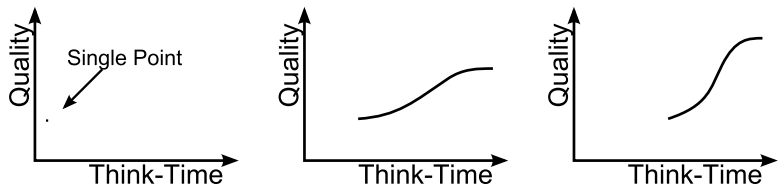

Figure 7.4: Quality of schedule chosen vs time taken to choose it. Three schedulers are shown, each with increasing abilities.

Figure 7.4 shows three quality vs think-time curves. Notice that the simplest scheduler has the smallest think-time, but also the lowest quality. As abilities are added, think time increases, as well as the achievable quality.

Static scheduling in toolchain: One dimension not indicated is whether this think-time occurs during the run, or in advance inside the toolchain. Some combinations of application

plus hardware have high predictability independent of the shape-control values within the input data. In this case, the toolchain can implement the think portion of the scheduler. It does a search for the best task boundaries, during which is does a sub-search for each candidate task-boundary choice. In the sub-search for a given set of task-choices, it looks for the best scheduling decisions by predicting the communication patterns resulting from a candidate scheduling and the interaction with the hardware communication paths and hardware schedulers. Once satisfied, it generates code that implements the decisions during the run. In this case, the think-time during the run is effectively just the time to look up what was pre-decided.

The best possible scheduler: The best possible scheduler is one that has perfect predictability of everything relevant, and performs all the decisions in advance. This actually is fairly common. For example, it exists for a compiler scheduling instructions onto a simple 5-stage pipeline that has only a single level of main memory and no cache. This occurs in micro-controllers in the embedded segment. However, it relies on the uniform task-size (one instruction), the fixed data consumed by the task (two registers), the uniform execution time of tasks (the cycles of an instruction), and the simple communication hardware (direct connection to memory processor with uniform access time).

Prediction vs complexity: For hardware with greater complexity, even just a single level of cache, the practicality of perfect prediction quickly disappears. As a result, the behavior of parallel code on current multi-core, and projected future, hardware cannot be well predicted for very far into the future. Prediction accuracy and distance into the future decreases and complexity of predictor increases for applications with input-dependent work-size, and/or input-dependent data-size, both of which are common. In this situation, the scheduler must perform search for highest quality schedule during the run.

The complexity of prediction was illustrated by the quality vs think-time curves in Figure 7.4. To get the effect of the think-time on delivered communication, those have to be combined with the communication curves from the previous sub-section.

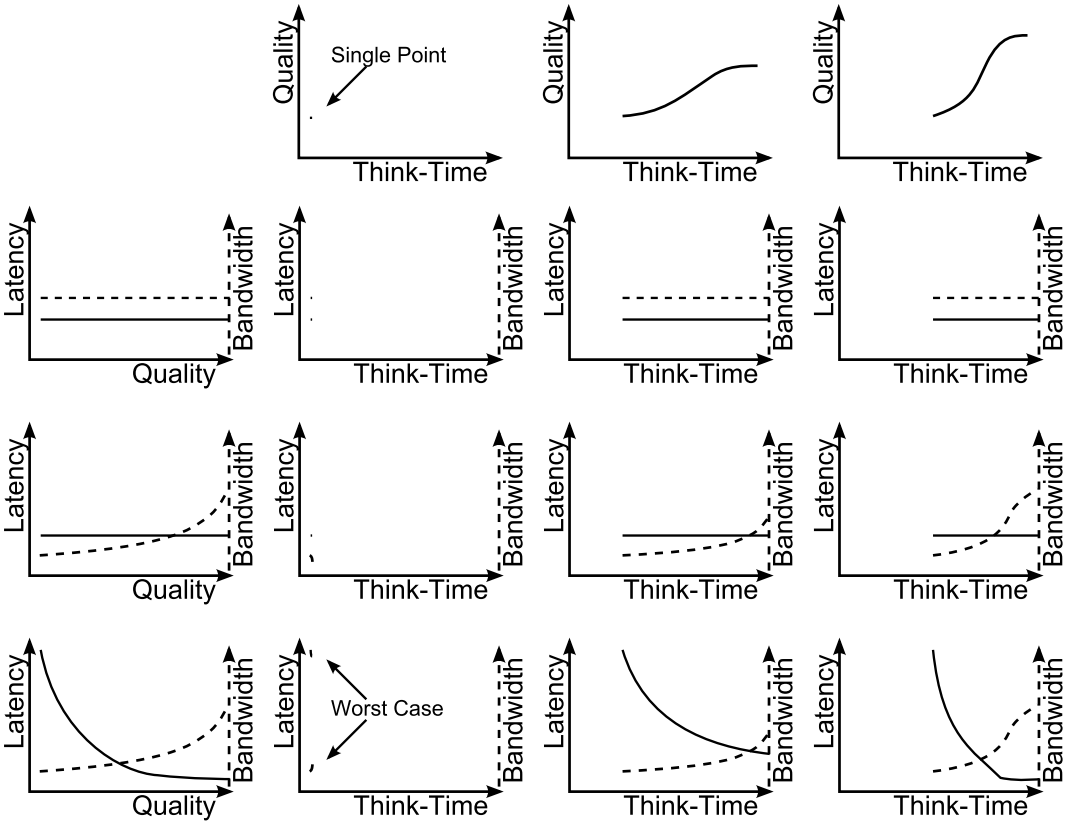

Figure 7.5: Data-Comm characteristics vs think-time, when comm-vs-quality curves are combined with quality-vs-think-time curves. 3 schedulers times 3 hardware-app combinations yields 9 curves.

Figure 7.5 Shows what happens when communication curves are combined with the quality curves. The result is the communication delivered to the task, as a function of how long the scheduler spent deciding the schedule.

Note the various implications from these curves: (top row) for simple combinations, the high think-time of schedulers with predictive abilities is counter-productive; (middle row) adding predictive abilities gives better communication at the cost of added think-time; (bottom row) for

some combinations, the improvement in communication characteristics can be dramatic, given enough predictive abilities and enough think-time, which matters the most for tasks with large data sizes; (all rows) when the data in the task is small, only latency matters to communication.

7.4.3 Iteration Time

This subsection considers the effect, on the time of an iteration, of both scheduling quality and the think-time required to get it. The next subsection will go on to consider how iteration time affects overall running time.

Curves are given for the communication time resulting from applying the communication characteristics to the data in the task. The think time is added to the resulting communication time, and that sum is added to the computation time and the return status time. Then, on the same axes, is plotted the total iteration time as a function of think-time.

Along the top, from left to right, the scheduler adds prediction abilities. Along the left edge, from top to bottom are the communication-vs-quality curves for a number of application-onto-hardware combinations. In the center are the resulting round-trip-time-vs-think-time curves.

The most important feature is the minima seen in the curves in the lower right. What's happening is that the think time grows larger than the communication and work time. If the scheduler were to continue thinking, it will only make the total iteration time worse.

Conclusions from curves: The most important conclusion is that think time eventually dominates improvements in communication gained from that extra thinking. For complex combinations, if quality is monotonic with think-time, there is always a minimum in the iteration-time-vs-think-time curve.

However, the iteration-time curves don't directly indicate the overall running time!

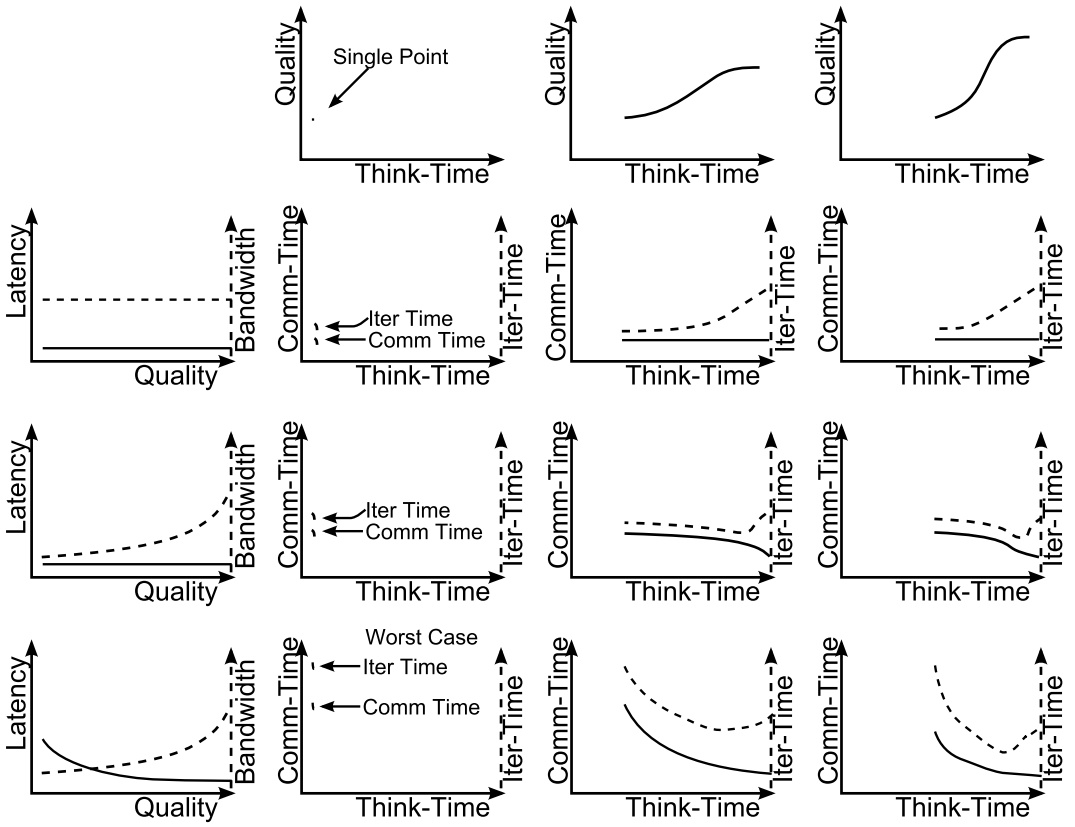

Figure 7.6: Plotted on the same axes are total communication time vs think-time and resulting total iteration time vs think-time. The task is the same for all, its data size is such that transport time equals latency for the worst-case, which is in the lower left corner. The work time is equal to the best-case latency.

Additional dimensions of interaction have to be added to translate round-trip iteration time into running-time, which will be done in the next sub-section.

7.5 Total Running Time vs Iteration Time

The iteration time is only part of the story. Iterations overlap, and it is the overlap that sets the total running time.

The overlap is constrained by two things: the task dependency graph, and hardware resource constraints. The task dependency graph sets communication between tasks, which

constrains the order of their iterations. The hardware adds limitations on overlap of iterations, and size of task.

Elaborating, the dependency graph determines which tasks produce data and which receive it. So, iterations for producers must complete before iterations for receivers can begin. In practice, multiple levels of hardware exist, and the processing elements can be virtualized, so partial overlap of producer and consumer iterations can be achieved, which will be detailed in a later subsection.

The hardware limits on overlap of iterations are due to a limited number of processing elements. This actually only constrains how many iterations are in the work phase at the same time. Some hardware also limits how many iterations can overlap the communication phase, due to limited communication resources. The data-size of a task is limited due to finite physical memory size, for example in distributed memory hardware[15].

Figure 7.7 illustrates a set of loop iterations mapped on to a simple task graph.

Two additional factors affect the overlap pattern: the choice of task boundaries, and the virtualization of hardware resources. Resizing a task changes the number of tasks. This, of course, changes the task-dependency-graph. It doesn't change the types of the tasks, so the nature of the dependencies stays the same, but it changes the number of a given type, so the size of the graph changes.

Resizing also affects the communication: the amount inside a single iteration, and the total amount for the run. It therefore affects the comm-vs-think-time curve shape inside each iteration.

Virtualizing the processing elements makes the scheduler have more physical elements available. As a result, multiple tasks can be scheduled to a single physical processing element. The virtualization will switch the physical processor among the tasks, animating first one then

[15]Limited local memory is a major constraint in Cell BE scheduling, for example.

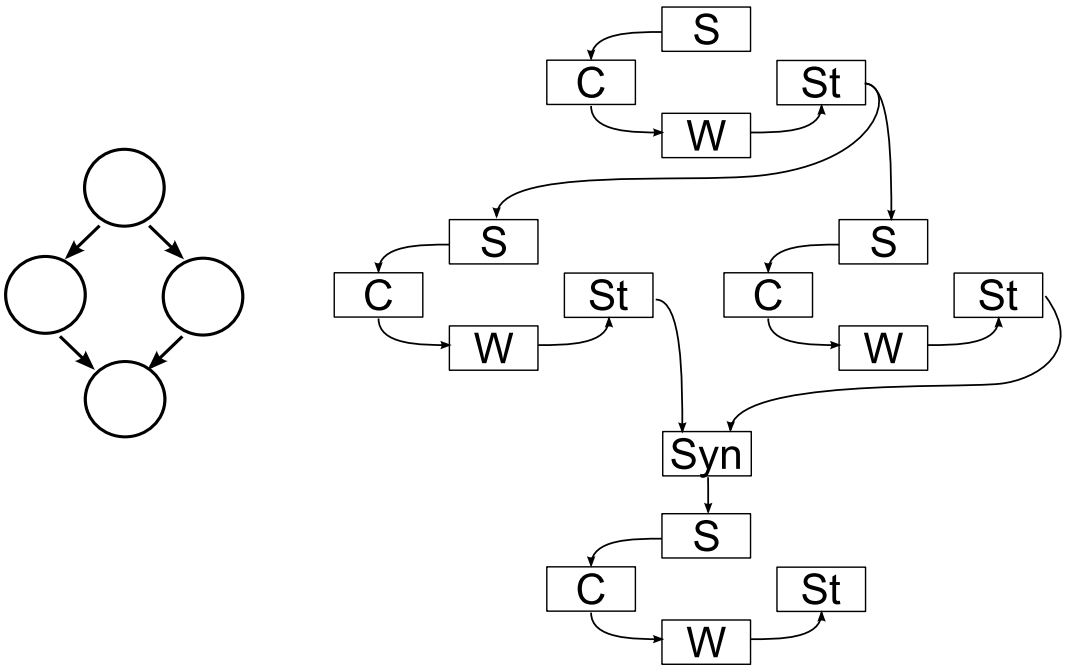

Figure 7.7: Scheduling loop iterations mapped onto a simple task-graph. Each iteration-box represents the time of that iteration-phase. The running time of the full application is thus the longest path from the starting schedule box to the final status box (assuming no virtualization underneath). The bottom-most iteration's schedule box must wait for both status communications before starting. The wait-time between the arrival of one status and the arrival of the other is represented by the Syn box.

another.

Virtualizing is useful when there is a separate processor for communication, such as DMA or a pre-fetcher. Then one iteration does its communication phase while another is in its work phase. Both tasks reside on the same physical processing element, but one uses the communication processor while the other uses the work processor.

This virtualization is an example of inserting an extra level of animator. The top level is the execution-model animator, which schedules to virtual nodes. The virtual nodes are the inserted animators, and schedule to the physical communication processor and work processor. The physical processors are then another level of animator.

7.5.1 Task dependency graph constraints

A propendent task sends its result to the dependent task. In the iteration overlap, this means that the earliest the second scheduling loop iteration can begin its scheduling phase is after completion of the status communication phase of the first iteration.

This is illustrated in Figure 7.8, showing the map of iterations onto timelines of two cores. Core 1 is assigned tasks 1, 2, and 4, while Core 2 is assigned task 3. It can be seen how communication time between the cores affects start times of the iterations.

Scheduler think-time onto cores: In general the scheduling think-time can be assigned to cores in various ways. One choice would be to perform all scheduling on one core, another, shown here, is for each core to independently perform scheduling. Here, the cores exchange status information at the end of each iteration either performs.

Notice that the status box can be multiple-valued. Here, St1 and St3 have a short time for on-core comm, and longer for inter-core comm. This affects the overlap pattern of the iterations onto the cores.

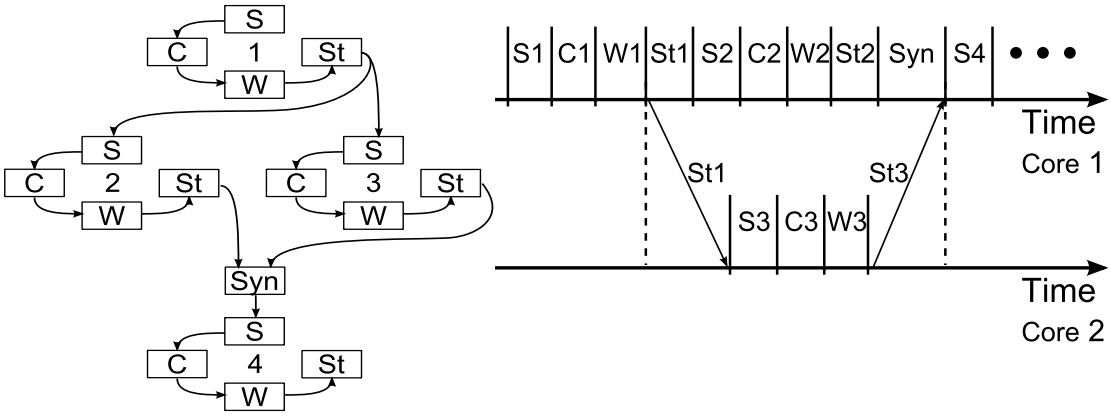

Figure 7.8: Scheduling loop iterations mapped onto the timelines of two cores. Iteration 1 is propendent of 2 and 3, and so communicates its result to them. Here, scheduling runs on both cores, but on Core 2 is only shown for the iteration in it receives a task. Status is communicated between cores, but only shown where it has an effect.

Width of dependency graph: An important feature of a task dependency graph, such as that in Figure 7.9, is the width at any given horizontal slicing. This width limits the number of tasks that can overlap within the scheduler's timeline.

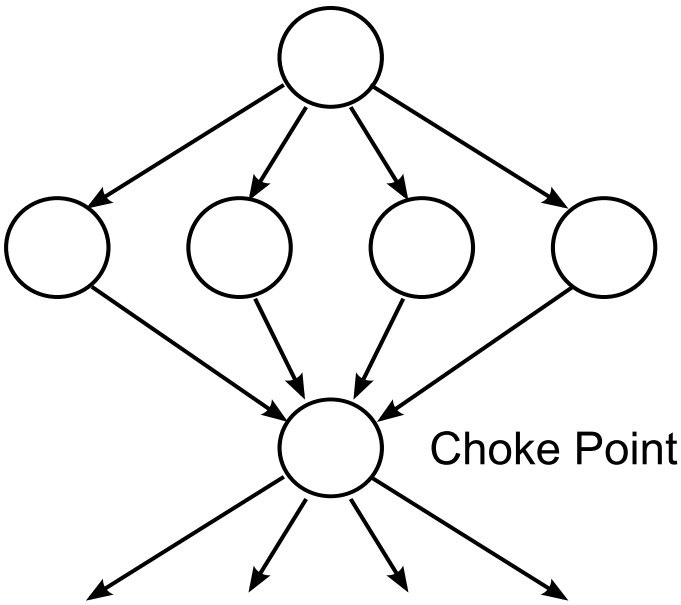

Figure 7.9: A dependency graph, with a choke point. The scheduler should attempt to minimize the choke-point's impact.

Places that the width is smaller than the number of physical processing elements, such as that labelled "choke point", will leave processing elements idle for a time. A high quality scheduler will be able to recognize such choke points and search for a way to start the key task as early as possible, especially if it is a long-running task. It will also attempt to run that task in a central place to reduce communication times.

7.5.2 Hardware resource limitations

Hardware limitations add constraints. A good example is limited size of intermediate memory, such as cache, scratchpad RAM, or register set. The scheduler should choose task boundaries such that the data-size is smaller than the available memory. If it can't make the

task size small enough, some larger memory will have to hold the data and communication take place back and forth. It should account for the effect of this increased communication in its placement choice.

Other hardware that constrains may include: prefetchers, DMA engines, memory banks, and so on. In general, any resource that is limited and can run out independently of others will place constraints on overlap of iterations.

7.5.3 Communication

As noted, the nature of the application determines how total communication volume grows as task data-size decreases. This affects individual iterations, by modifying the communication-vs-quality curve, which in turn affects the iteration shapes, which combines with the constraints to affect overlap and hence total running time. However, physical communication pattern in the hardware and communication resource limitations often don't introduce constraints, but only affect the quality curve in the iteration. However, when applying the model to the communication scheduler, the physical paths do indeed introduce additional constraints. But, the constraints depend upon previous scheduling choices, so each choice modifies the constraints for future scheduling choices.

7.5.4 Effect of choice of task boundaries

Tasks can be divided, coalesced together, or the application-processor definitions modified to change the code in tasks. Each change modifies the task-dependency-graph and also likely interacts with the constraints imposed by hardware resource limitations.

For example, changing task data-size goes hand-in-hand with limited cache or scratch-pad memory.

7.5.5 Virtualizing Resources

The most common form of virtualizing hardware resources is double, or triple, buffering. Each buffer represents a virtual physical animator. Communication takes place with one while the other is computing. Switching between them takes place in a lower-level animator that is local to the physical processing element.

Virtualizing the physical animator in this way increases overlap of scheduling loop iterations. It does this by increasing the number of processing elements to assign tasks to. When one the virtual processing elements goes idle, the physical animator switches to a different virtual animator. Normally, the idleness is caused by waiting for communication, especially in a lower-level of the physical animator hierarchy. This communication is invisible to the scheduler that assigned to the virtual animator. Hence, the physical animator below gets more of its time spent computing, less idle. The effect is that the group of tasks assigned to the virtual animators complete faster than if they were assigned in sequence to the underlying physical animator.

The level whose processor assigns tasks to the virtual animators (virtual processing elements) does not know they are virtual! One effect is to reduce predictability of completion time of an assigned task. The model of computation rate becomes more complex, and now includes application characteristics.

When to virtualize: Such buffering is normally only profitable when the task data-size can be modified to fit the buffer size. The choice to use such buffering must be done with an eye towards the effect on total communication, and how that will affect the overlap of the tasks. When communication within a loop iteration is larger than work-time, and making tasks smaller increases total communication, then multi-buffering is a net loss.

It would be good for specialization to include a means to quickly predict whether multi-buffering will be profitable, and let the scheduler turn it off an on at will.

Relating virtualizing physical animator to the levels of hierarchy: Virtualizing the physical processors adds a level of animator. The extra level breaks up the task assigned to a virtual processor into multiple smaller tasks and schedules the smaller ones onto the underlying physical processor. Two or more virtual processors exist on a single physical processor node, so sub-tasks from one virtual processor intermix with sub-tasks from the other virtual processors.

Events within a task trigger the virtual processor to suspend and switch the physical over to a different virtual processor. A triggering event would be something like the task initiating a communication. Such an event is what ends one sub-task, and causes a sub-task from a different virtual processor to be scheduled onto the physical processor. When the switch-triggering event clears, for example the communication completes, that is viewed as making a new mini-task ready. This behavior is that of classic OS threads.

Definition of Virtual: Viewed from this model, the word Virtual is defined as a software processor having the same interface and behavior as a physical one. That software processor may be animated by a physical processor having a different interface and behavior, but not always.

7.5.6 Conclusions from Thought Experiments

The job of the scheduler in a level of animator is to perform a search, to find the optima of the goal measure. The scheduler's think-time is inside a loop, so the longer it thinks, the longer becomes the iteration time of the loop. The iterations overlap each other, according to task-dependency-graph and hardware constraints. So iteration time affects total running time in some complex way related to both application characteristics and hardware patterns.

Let's say the goal is shortest total time of a run, and look at the implications for the perfect scheduler.

The perfect scheduler should predict: the tasks to be produced in the future, the time to communicate their data, time to do the work, and time to communicate status back. With these predictions, its job is to search through all valid changes to task-boundaries, in combination with all valid assignments of tasks to processors, including predictions of interactions of schedulers at all levels of the animator hierarchy, and find the assignment that gives the best goal-measure.

The goal of the search is minimum running time, so for communication bound applications, the goal becomes to reduce total communication. In contrast, for compute-bound, the goal becomes to hide communication, by overlapping it with work.

For communication-bound, the scheduler is free to use up extra think-time, by filling the idle processor time with thinking. But for compute-bound, it should use only enough think-time to avoid having idle time.[16]

Hiding communication means virtualizing the physical animator, by having task buffers in memory close to the processing elements. One is filled by an independent communication processor, like DMA or a prefetcher, while application work is performed out of another buffer.

The job of specialization will be to insert the scheduler that comes as close as possible to the perfect one, given the limited think-time imposed by task size and hardware. This conclusion will be explored more detail in the next section.

7.6 Relation Between Scheduler, Application, and Hardware

The previous section explored the consequences of a scheduler's ability to search for good scheduling choices. It showed that a number of scheduler abilities are important to this search.

[16] assuming the same processor does both scheduling think-time and the application work

In this section, the relationship is explored between scheduler abilities, the characteristics of the application being animated and the hardware doing the animating.

The scheduler's implementation only acts as an enabler for predictive abilities. The application and hardware must inherently have predictability, and a predictive model must be extracted from the code and hardware and made available to the animator. Only if all three aspects are present does the scheduler exhibit predictive abilities during its search.

7.6.1 Animator Hierarchy

Before proceeding, an important phenomenon should be explored a bit. The physical animator often has a hierarchy of processor levels. The execution model's animator should be aware of this hardware hierarchy. It is often profitable to match levels of execution model animator to levels of hardware animator.

The interactions between levels in the hierarchy of animators (physical and software) should be modelled. One level's scheduler is affected by lower level's schedulers. A given level sees the behaviors of lower levels as a non-stationary probability distribution.

For example, take a multicore chip with coherent caches. The execution model's target animator sees the existence of cores, where a core is defined as one execution pipeline with one level of cache. It may have multiple virtual cores implemented in hardware, via SMT (symmetric multi-threading) or other hardware multi-threading.

The execution model (ExeMo) animator assigns a task to a core. But that task is then broken into a bunch of little tasks by the hardware-scheduler in the cache! No communication internal to a task is seen by the scheduler that assigned that task. Only the communication to get that task to its assigned location is seen. All the side-effects that happen via communicating with persistent memory processors are internal to the task.

One way to view a task is as a single execution-model processor, for an execution-

model that is inserted during the specialization process. When looking at things this way, there is a hierarchy of execution models, one inserted for each level of hardware. Hence, the behaviors of lower hardware levels are invisible to a given execution-model animator. It has to treat their effects statistically, even though there are deterministic, calculable interactions among scheduling decisions in different levels.

It will be seen that making the schedulers, in the various levels of execution model, aware of each other has benefits. It reduces the complexity of search that each scheduler performs, and it improves the predictions that each level makes. However, at the lower levels, the amount of think-time available to the scheduler decreases, and so less and less prediction is available to the the lower-level schedulers.

7.6.2 The Scheduler Abilities

This subsection dives into a bit more detail on the particular scheduler abilities. There are several abilities that improve the scheduler's effectiveness at finding both the best choice of task boundaries, and best assignment of them to lower-level animators.

Predict future scheduling decisions: Sometimes the locally best scheduling decision is globally non-optimal. Hence, a scheduler that cannot predict future scheduling decisions is following a greedy algorithm. The opposite is also true, a scheduling decision that is the best globally is only guaranteed the best locally if all the other decisions are also the best globally. Any of those globally best may be an extra-poor choice when mixed in with sub-optimal decisions.

Only if the scheduler can predict all future scheduling decisions, can it search globally for the set of all optimal decisions, and only then can it find the globally best for a particular scheduling-loop iteration. This informally observed behavior is consistent with scheduling being an NP-hard search problem.

Predict task data-size: When the scheduler can predict the data of a task, then it satisfies one necessary condition for predicting communication time of a particular scheduling-loop iteration.

Predict data location: If the scheduler can predict which places in the memory hierarchy hold each bit of data in a task, then it knows the communication end-points. These can be used with ability to predict communication resource usage.

Predict communication resource usage: If the scheduler can predict the behavior of the communication hardware, in combination with knowing the communication end-points and sizes, then it can predict the communication parameters for a given scheduling assignment.

Predict communication parameters: If the scheduler knows the state of the communication hardware, and knows the characteristics of its communication paths, and can predict future uses of those communication paths, then it can predict the latency and bandwidth delivered to a given task. This predicted value changes for each candidate assignment of the task. The search through the possible assignments is one dimension of the scheduler's overall search for best scheduling choice.

The best possible prediction depends on also having prediction of future scheduling decisions, and prediction of communication hardware behavior. Even without these, some degree of communication prediction can be performed, and is beneficial, as seen in the thought experiments. Reduced accuracy of any of these predictions simply lowers the quality of scheduling decisions possible.

Predict task work time: If the scheduler can predict how long a given task will take to execute on the assigned processing elements, then it can predict the time of the work-phase of

the iteration. For schedulers that employ virtual processing elements, this value tells it at what point the physical animator will switch to a different virtual processing element.

Summary of practical prediction abilities: In practice, if the scheduler can predict the location of data needed for work, then it knows the end-points of communications. If it can also predict the amount of data needed, then it can use a network model to predict the amount of time to complete the communication. If it can also predict the amount of work, then it can use a computation model to predict the time to complete the computation. If it has the task-dependency graph, then it can put these all together to predict iteration-overlap and effect of candidate choices on overall running time.

Ability to resize work in a task: Another valuable ability is resizing the work in a task. A task consists of particular code, particular shape-control values, and the flow-through data. Together, these determine the size of the equivalent circuit, which is the amount of work to be performed. [17]

Hence, two dimensions exist for changing the work in a task: change code, or change shape-control/flow-through data.

The least complex way to modify the amount of work is to change the shape-control data. This normally implies an accompanying change in flow-through data. For example, modifying the iteration bounds in the matrix-multiply loop nest is equivalent to setting boundaries in iteration space. Each point in iteration space accesses a particular combination of cells from the two input matrices. Hence, changing the shape-control data changes the flow-through data accessed by the task.

If the scheduler can resize, then it can search for a size that results in coming closest to

[17] Recall from Chapter 6 on the Basic Model that the equivalent circuit continues expanding as computation moves forward. Even if the code is signal-processing type, which represents a fixed set of operations repeatedly carried out, each application of those operations to input data causes creation of more equivalent circuit to be added on, when performed on a Von Neumann type processor.

the goal. For example, in matrix multiply, sub-block size would be chosen as large as possible, to minimize communication, but small enough to keep all computation elements busy.[18]

This is especially important because the complexity of applications in terms of total data to communicate is often non-linear vs the size of task. Matrix multiply is a good example. The more sub-blocks a matrix is broken into, the more total communication to processing elements must take place. This is easily seen, because a single sub-block in the left matrix must be paired with each sub-block a right row.[19] The more sub-blocks, the more pairings, and the same left sub-block must be sent through the network for each pairing. So total communication volume goes up as the number of sub-blocks in a row.

Ability to change task code: Modifying the code within tasks can be done for two reasons: either to change the amount of work, or to affect schedulers of lower-level animators.

The task code equals specifications of the application processors. If only a single level of physical animator hierarchy exists, then rearranging the task code only affects the amount of work in the tasks. However, on hardware with many levels of physical animator, the pattern of code inside a task affects the behavior of lower-level animators.

The best example is loop interchange for a loop nest inside a single task-type's code.[20] That interchange has no effect on the amount of work in the task, but it does affect the cache behavior! The cache is a memory processor that receives communications from the main "work" processor, and the order of addresses affects the scheduling decisions inside it. Some orderings interact poorly with the cache's internal scheduling algorithm and its lower-level animators.

As a result, the ability to modify the code of a task is a very important ability, and

[18] Block size doesn't normally divide evenly onto the number of physical processing elements, so various approaches to tapering off size over time, and so forth, are taken in real schedulers, either embedded in the Language's animator [63] or hand-coded in the application.

[19] The column-number of the left element equals the row in the right matrix it ends up being multiplied by, over the course of many left-row times right-column products.

[20] A task-type is defined by the code inside a task instance. All task instances with the same code are members of the same task-type.

also a difficult one for the scheduler to make use of. Sequential compiler optimization is the process of modifying code such that the various levels of hardware animator interact favorably, and its complexity demonstrates the difficulty of the problem.

7.6.3 Task Characteristics Enable or Disable Scheduler Abilities

A scheduler may be implemented with the ability to predict work-time of a task, but it can only do so if the application enables that ability.

For example, for an application like matrix multiply, the work-time can be accurately predicted. Once the shape control values are known, the amount of work is fixed. In contrast, for an NP complete problem, the predictability of work-time depends on the nature of the input data (which is almost all shape-control data). Given the same code, same size of input graph, and same number of nodes and edges, some data sets have predictable amount of work and others are inherently unpredictable (unless P equals NP, which has a lot of evidence against it).

Hence, NP complete problems can effectively disable a scheduler's prediction ability. Consider what happens for a scheduler that has work-prediction implemented. For NP complete problems, the best the scheduler can do is somehow get a custom widget that can analyze a given input graph. For some graphs it will return a distribution of execution times, but for other inputs, no accurate distribution exists! In this situation, the work-prediction implemented in the scheduler is effectively turned off by the application.

The characteristics of applications that enable or disable key scheduler abilities are:

- predictability of task's work amount
- predictability of task's flow-through data footprint
- predictability of task's dependencies and propendencies[21]

[21] Dependent and propendent are defined as: a dependent task receives communication from its propendent

7.6.4 Hardware Characteristics Enable or Disable Scheduler Abilities

The things in the hardware that affect scheduler abilities are:

- levels of hardware below the level the scheduler is in
- schedulers hard-wired into the hardware
- communication patterns in the wires
- sizes of memory arrays

In general, having hardware animators at lower levels reduces the predictability of the communication and execution times of a given task. The scheduling decision for the task models the processing element it is assigned to. The more internal processors in it, the more complex its behavior, and the less predictable it is.

Schedulers with hard-wired specifications generally don't communicate and cooperate with higher-level processors. This makes the schedulers in the higher-levels blind to behavior in lower levels, which reduces predictability, and makes the search at higher levels more complex.

Patterns in the communication wires affect the model that higher levels have of the communication characteristics. Some patterns make it easier to predict communication parameters than others.

The sizes of memory arrays at a given level of the hardware hierarchy affect the maximum size of tasks scheduled at that level. This affects the ability of the scheduler at that level to resize tasks. It constrains the choices of task size.

task(s). An independent task has no propendents, but may be a propendent itself, sending communications to its dependent tasks. A dependent receives, while a propendent sends.

7.6.5 Extract Task and Hardware Characteristics for the Animator

In general, the scheduler needs to be supplied with the task dependency graph, plus models with which to predict a task's amount of work and amount of data. How to extract these from the application and supply them to the scheduler is part of the subject of specialization, covered in the next chapter.

Either the scheduler at a particular level of the animator hierarchy has to be implemented with the hardware characteristics of other levels in mind, or else those characteristics have to be modelled and given to the animator. Normally, only the levels lower in the animator hierarchy have to be modelled. However, to reduce complexity of prediction and search, each level should negotiate its decision-making with the others.

7.6.6 Hardware Interactions with Software Affect Scheduler Decisions

In addition to affecting the scheduler's abilities, parameters of the scheduler's search are also set by the detailed interactions between application and hardware. These parameters values are used within the scheduler's decisions.

For example, the hardware sets the communication paths, and past scheduling decisions set the current and near future usage of those paths. Given that context, the predicted communication characteristics combine with amount of data in the candidate task, to set the task-communication-time. This is the time of one phase of the iteration, and so fixes one point of possible overlap.

Next, the work-rate is set by the hardware, plus past and future scheduling decisions if the hardware has multiple levels that can interact. This combines with the work-amount in the task to set the work-time for the iteration. This fixes another point of possible overlap.

The status feedback is then set mainly by the hardware, but influenced by past and

future scheduling decisions.

These three values for the iteration phases constrain the scheduler think-time. The task dependencies plus animator virtualization set the possible overlap points. These determine whether the work-processor will have any idle time to burn up in extra scheduler think-time. If not, these three phase-times limit the amount of productive think-time the scheduler can perform.

Consider the case when far down in the animator hierarchy, close to individual transistor switching times. For example, in the pipeline of a work-processors. Here, the three phase-times are so short that the scheduler has no time available to perform predictions, nor even searches. In such cases, scheduling decisions must be decided in advance and encoded into the executable.

Pre-computing scheduling decisions is normally the job of the compiler. Part of the optimization process is performing the search for good scheduling decisions.

In contrast, when high up in the animator hierarchy, for example, in grid computing when scheduling tasks over planetary-scale wide-area-networks, the scheduler has a great deal of time to predict behavior and perform a detailed search.

This demonstrates that the best scheduler implementation is highly dependent on not only the hardware, and not only the software, but the detailed interaction of the two.

7.6.7 Choose Scheduler by Both Task and HW Characteristics

So far, it is clear that multiple scheduler implementations should be available. Each implements a different combination of abilities. The tasks and hardware combine to enable a combination of abilities. So, the only scheduler implementations that should be considered are the ones whose combination of abilities is enabled.

Now, that's not the full story. As seen in the previous subsection, the maximum

productive think-time also limits which scheduler implementation is viable. In the limit, only the no-think scheduler is viable, despite what abilities the task and hardware might enable.

When the scheduler matches the code plus hardware characteristics, and those characteristics enable enough think-time to take advantage of all the enabled abilities, then the highest quality schedule is achievable, without inefficiency.

In the case that a scheduler with many abilities is being used, it likely will require much more think-time than is productive before it finds the best decision it is capable of. So, while it is performing the search it should also be estimating the amount of think-time it has left. It should be keeping the best candidate decision found so far, and when the think-time runs out, use that best so far decision.

Conclusions for Specialization: The conclusions for specialization are that it should be aware each level of animator hierarchy, and the hardware characteristics at them. For each level that it inserts a scheduler implementation, it should attempt to identify phases in the application that have a characteristic working-set of task-types. And for each application-phase, estimate the minimum task communication time, work time, and status time. With these, it can calculate the maximum think-time available. It should then choose a scheduler implementation to insert that has as many abilities implemented as are enabled, and that can achieve reasonable quality within the allotted think-time.

Sequoia partially addresses the issue of choosing the best scheduler implementation by forcing the application to include a hand-coded scheduler for each machine-architecture the application can be run on. In effect, the human programmer is expected to balance the capabilities placed into the scheduler against the characteristics of the hardware, given the characteristics of the application. Notice that in Sequoia the tools haven't been supplied with the information they need to re-size the tasks, nor is the human programmer given any means of gaining insight

into which abilities are enabled, nor how much think-time is productive.

Insights applied to industry approaches: Another way this section's insights might be applied is in recognizing that any two of three things can be fixed, and the third chosen to balance with the other two. The three things are: the task characteristics, hardware characteristics, and scheduler implementation.

If the task-types and hardware are fixed, which is the case for general-purpose computing, then the scheduler implementation has to be adjusted to balance with them. This should take place during the specialization process.

For custom hardware solutions, typically, the scheduler implementation and hardware are fixed. Those determine what kind of applications will run efficiently on that custom solution. Typically, the language is fixed in this case, and will have limited support for extracting prediction information. The custom hardware typically includes hard-wired scheduling logic.

Lastly, if a set of applications is chosen, along with a range of scheduler implementations, then hardware must be designed to accommodate those. An example of this case would be a company choosing to focus on signal processing, and then choosing an existing language. The language limits the scheduler implementations possible, and the class of applications fixes many task characteristics. The company would typically develop a custom toolchain that a-priori knows the application characteristics and they have scheduler techniques in mind. They then have to pick hardware that fits with those choices.

Chapter 8

Specialization

This chapter is about the process, by which source code is made efficient on target hardware. Whether done by hand or via automated tools, the process rests on the same principles, illustrated with the Holistic Model. This chapter discusses the process and ties it to the model.

In general, specialization is a collection of techniques, applied in the infrastructure at various points in an application's lifetime. The effect each has can be understood in terms of the Holistic Model. This chapter first describes specialization, stating its goals, and revealing structure. It then discusses some existing specialization techniques, and finally uses the Holistic Model to motivate future techniques for better portability of parallel code.

8.1 What is specialization?

In a nutshell, specialization is a process, which makes application code fit with target hardware. The purpose is improving some goal measure, usually performance, or a combination of performance and energy efficiency. The process is some collection of techniques that can vary

widely depending upon the rest of the software infrastructure and the hardware being specialized to, and even the properties of input data. The process can take place at any one point in the application lifetime or it can be spread out and performed in many bits and pieces over that lifetime.

The techniques used to improve the code can include manipulation, translation, injection, and linking. Manipulation is normally driven by communication patterns in the hardware, and effectively changes scheduling decisions that were made by the programmer, so that the new ones cause better communication patterns.[1] Translation casts the code onto a different execution model, followed by injection of an animator for it, or down-stream linking to an animator.[2]

8.1.1 The Four Things at the Heart of Specialization

The four things inherent in specialization are information extraction, modelling, modification, and search. First, information is extracted from the application, from the hardware, and sometimes from the properties of input data. Then, the hardware is modelled, and the application is modelled, so that patterns generated by the application can be predicted, and the consequences of presenting those patterns to the hardware predicted. Then, a search takes place wherein projected modifications to the application are tested. The post-modification patterns are predicted, and the effect those have on hardware is compared to previous.

All specialization techniques can be viewed as some combination of these four. Some may only perform one type of information extraction, or might perform a fixed modification without modelling nor search.

In practice, specialization for parallel software is currently just a collection of techniques, no broad-based infrastructure exists to exploit the patterns within it. Each combination

[1] Traditional compiler optimization falls in this category.
[2] This is the base function of a compiler.

of target machine and development tools has its own collection, no segment-wide organization exists. However, in the future, new languages should be designed to support a wide variety of specialization techniques, and infrastructure should become standard that is aligned with the information extraction, modelling, transform, and search at the heart of specialization.

8.2 When is Specialization Performed?

The techniques can be applied at many different points in the application's lifetime: during the development cycle; after testing freezes source but before distributing; just after distribution while installing; at the start of a run; interrupting the run periodically; in parallel with the run; after the run (profile-driven). Each point has implications on the quality of specialization, and practical implications, which in turn affect adoptability in a given software segment and project size.

Many existing techniques perform specialization at multiple points in an application's lifetime. For example, auto-tuners are currently inserted pre-distribution. They contain a number of code variants that each were specialized pre-distribution. During initialization of a run, or the run itself, the input data becomes known, and the auto-tuner picks from among the code variants. This choosing is equivalent to swapping in different manipulations of the code.

8.2.1 Choosing When in Application Lifetime to Specialize

The choice of which point or points at which to specialize affects adoptability: it may force changes in development practices, may affect level of reliability of delivered code, and affects the level of performance achieved.

More importantly, the infrastructure surrounding an application puts constraints on which techniques can be applied at what points. These constraints are due to information about

the hardware, the amount of time available, and knowledge of the input data.

Pre-distribution: The most time is available before distribution. Complex searches are possible, and the most sophisticated, long-running techniques can be employed, even running for days.

A major advantage of specializing before distribution is that programmers can interact in the process. Once software ships, only automated specialization can be performed.

However, there is a tradeoff: if full hardware knowledge is available, then specialization becomes logistically challenging and likely expensive. When programmers are involved in specializing, they have a large number of targets they have to spend time on. And, it is logistically challenging to manage the many versions of code, the distribution of multiple executable versions, and the testing and verification of all the different executables for all the different targets.

In contrast, if specialization is only targeted to general hardware classes, in order to reduce the effort and logistics, then the manipulations are limited, and down-stream specialization will likely be needed to win back the lost opportunity for performance.

During installation: When an executable is installed onto hardware, a fair amount of time is still available to perform manipulations, and full hardware knowledge is available. This makes it a good choice for automated specialization, perhaps as a second step in the process.

The main limitation is that the programmer is not available to take part in specialization. The second limitation is time: pre-distribution can run for days, but install-time specialization has only minutes for laptop and desktop, and seconds for mobile devices. But server-installed software can take much longer.

One complication is that the installation process must invoke the specialization. This

may require a change in the software stack. However, on some OSs, installation scripts are already possible to do this.

At initialization: This is the first point at which input data can be known. Unfortunately, only a matter of seconds are available during start-up of a run, so initialization-time specialization typically requires up-stream processing that has pre-simplified the processing needed. No changes to the software stack, nor separate install scripts are needed. For example, auto tuners often run during initialization, for programs that process very large datasets.

During the run: Only a small amount of code manipulation is profitable when done during a run. The way to improve this is pre-computing possibilities that might be encountered. This reduces the specialization during the run to just identifying a hardware pattern then substituting in the pre-computed manipulation. Multi-versioned kernels is an example of this technique.

One advantage of this point is that statistics can be measured. This simplifies prediction: neither software nor hardware need be analyzed and modelled. However, the complex interactions among levels of hardware typically makes such statistics non-stationary, and measurements highly sensitive to small changes in scheduling decisions. An example would be StarPU [12], which gathers work-time statistics it uses to choose between CPU execution and GPU execution.

After a run: Post-run specialization is helpful to improve prediction for the scheduler. When prediction is too difficult to extract from the source and hardware, statistics can be gathered during runs, to build up a distribution for communication characteristics, work-time of tasks, data-size of tasks, iteration-overlap, and so on. Such statistics are, in general, non-stationary due to the detailed interactions between levels of scheduler, notably the cache's scheduler, but they may still be profitable to gather and use.

Runtime statistics gathering may be profitable when communication times are high (such as for GPUs involving transfer to different memory processor) or especially sensitive to choices (such as networks prone to congestion, which exponentially worsens characteristics).

8.2.2 Examples of Specialization at Various Lifetime Points

SEJITs: SEJITs [20] is a JIT-based approach, which specializes both pre-distribution and during the run. It provides a number of functions in libraries for the application to use, and does significant specialization of the library implementation before distributing it. They have full hardware knowledge when writing the library, and hand-specialize the kernels. During the run, the JIT links in the library for that hardware, and performs simple code manipulations such as in-lining code. One advantage is that it works with dynamic languages and ones that have higher-order functions.

The main disadvantage is that it does not have time to take into account very much application-specific information. The main specialization is performed on pre-defined library functions. This limits the type of applications it will be useful for. The small amount of specialization time available during a run constrains it to only apply simple manipulations and short, incomplete searches for optimal lower-level hardware scheduling choices.

VMS: The VMS approach specializes at three different points: before distribution (in the toolchain), at initialization (linking the animator in), and during creation of the software stack (implementing the VMS-core abstraction on the hardware). Because it starts at pre-distribution, it has no time constraints on the techniques that can be employed. However, the VMS approach limits the toolchain's information about the hardware, to just the general architecture class, which constrains the manipulations.

To alleviate this, VMS does allow down-stream specialization steps, so the lower levels

of hardware can be statically scheduled during installation, when ample time is also available for searching through manipulations. The pre-distribution specialization will have to supply the information needed by the install-time specializer, in order to get good quality static scheduling decisions. In general, all techniques available to JIT specializers are also available to VMS.

BLISS: BLISS has the fewest constraints on specialization techniques because it has full or near full information about the hardware before distribution, and the server approach enables long search times. The server can perform runs to gather profile information, and it can allow programmers to interact with the specialization tools. In addition, all the down-stream techniques of both JIT-based and VMS's morphable execution model are available for use within BLISS,

8.3 Where is Specialization Performed?

Specialization is performed in the infrastructure surrounding an application. Sometimes the creation of that infrastructure is part of the actual specialization, for example, hand-tuning kernels for a library-based specializer. Sometimes, tools in the infrastructure interact with a programmer to perform specialization. Sometimes, a specialization module is injected into the executable itself.

The life-point at which specialization is performed sets the place in the infrastructure.

One dimension of classifying specialization techniques is that some analyze code and others modify code.

8.4 Structure of Specialization

This section goes into detail on the information about application and hardware that is useful to specialization, and how code manipulations are driven by that information.

8.4.1 The Four Things at the Heart of Specialization

The four things inherent in specialization are information extraction, modelling, modification, and search. First, information is extracted from the application, from the hardware, and sometimes from the properties of input data. Then, the hardware is modelled, and the application is modelled, so that patterns generated by the application can be predicted, and the consequences of presenting those patterns to the hardware predicted. Then, a search takes place wherein projected modifications to the application are tested. The patterns generated post-modification are predicted, and the effect those have on hardware is compared to previous, directing the search.

All specialization techniques can be viewed as some combination of these four. Some may only perform one type of information extraction, or might perform a fixed modification without modelling nor search.

8.4.2 Specialization Is Related to Scheduling

In effect, all four of those things – information extraction, modelling, modification, and search – are related to scheduling in some way.

Extracted information relates to scheduling: The information extracted from the application includes boundaries of tasks, which the scheduler uses, and dependencies between tasks, which constrain scheduling choices, manipulators, which give the scheduler control over smaller-scale specializations during a run, and predictors, which scheduling uses during search for the

best choice.

Modelling is used by scheduler: Specialization constructs or has implemented into it a model of the hardware, and uses some of the information extracted from the application to model the patterns it generates, and model characteristics of the tasks. These models can be made available to the animator's scheduler, which uses them to improve the quality of its runtime decisions. The models can also be used by the specializer to make static scheduling decisions. Especially for lower levels of the hardware, the specializer uses the models to predict patterns of requests generated by the application, and predict the reaction the hardware's schedulers will have to those. The predicted effect then drives modifications of the static code inside a task, which equals a different set of static scheduling decisions.

Modifications relate to scheduling: The modifications are either rearranging code, which equals making static scheduling decisions, or inserting an animator, which has a scheduler inside, or inserting information for use by the animator's scheduler, or inserting a down-stream specialization module, which just delays doing one of the preceding. However, some specializer modifications affect data layout. These modifications, though are driven by the effects on the schedulers in the hardware, especially in the cache and network.

Search relates to scheduling: The search performed during specialization is a search through possible manipulations of the code.

An example of searching through code manipulations is an auto-tuner, which searches among alternative codings of a kernel to find the one with best performance on given input data. Each kernel equals an alternate static scheduling. The characteristics of the input data affect the patterns generated by a given kernel. Some data-kernel combinations generate requests that fit the hardware better than others.

Another example of search is optimization of the code inside a given task-type. Such optimization could be, for example, traditional sequential optimization. This amounts to trying different sequences of transforms, driven by some code-properties empirically discovered and encoded into the search as heuristics. The result of each candidate sequence is tested in a model of the hardware and the best kept, until some end-condition for the search is met. This amounts to searching through alternative static schedulings.

For higher-level structure, the search might be through alternative task-boundaries. For example, splitting a single complex kernel into multiple simpler kernels, and making each a separate task-type. This would make the code map better onto GPU hardware. Or the opposite, collecting multiple kernels together into a single complex task, to reduce inter-kernel communication, which would be better on a multi-core with large caches.

8.4.3 Extracting Information During Specialization

Specialization abstracts away details of the language, to extract information.

8.4.3.1 Extracting Prediction Information During Specialization

Specialization extracts prediction information to help the injected scheduler produce high quality decisions.

The language should be designed to facilitate. Perhaps the programmer is asked to encode functions that take a candidate task and return a distribution of work-amount, and/or data-size.

The predictions performed include pattern of communication-requests, the resulting latency and bandwidth delivered by the hardware, the work-time, status feedback time, scheduler think-time and hence the scheduling-loop iteration time. The overlap of iterations is also predicted. This overlap prediction requires the task-dependency-graph, the hardware-imposed

constraints, and a model of the virtual physical animator instances existing.

The previous chapter showed the importance of prediction information for communication-request patterns and how those interact with hardware patterns. The manipulation information is used by specialization that happens later in the application lifetime.

8.4.3.2 Extracting Task Dependency Graph

The schedulers at all levels require the dependencies between the tasks they schedule. For static decisions, the specializer itself uses the task-dependency-graph to make sure its transformations are valid.

Scheduling constraints are defined by the source-language, and so are related to the task graph. In fact, the constraints must be extracted first. They constrain the choice of task-boundaries, and the inter-task constraints become the task-dependency-graph. Scheduling constraints also set the bounds on manipulations of code inside of a task.

8.4.4 Modelling Application Behavior

The task-dependency-graph determines the overlap of scheduling loop iterations, from which communication-requests, demands for computation, and so forth are determined. The iteration-graph models application request generation. This will drive manipulations of task-boundaries to ensure enough overlap is available to fit the hardware and the number of virtual processing nodes.

Predictors of task work-amount and data-size are also models of aspects of the application.

8.4.5 Modelling Hardware Behavior

The injected animator is often implemented separately for each hardware target. Details of communication network, memory hierarchy, and so forth can be placed directly into the animator code. This is a form of giving a model of the hardware to the animator, for use in predicting effect of application upon hardware. However, the code often embodies heuristics that the programmer has identified. The specializer may not calculate any predictions, nor perform search, but rather the programmer has performed them, and simply includes a number of choices in the code.

8.4.6 Modifying Code During Specialization

The specialization manipulations must preserve both the interface to the application and the observable behavior. However, they may change the internal processor specifications, processor hierarchy, and especially the target animator.

A common technique is to insert scheduling that is designed to use the extracted information. The inserted scheduling may just implement static decisions made in the toolchain, or may be a dynamic scheduler, located in the target animator. Or be hard-coded scheduling implemented by a programmer using a profile-driven tool to specialize by hand.

Specialization's goal measure is most affected by communication patterns, whose endpoints are requested by the animator's scheduler. Those requests cause the hardware to engage in actual communication.

Specialization's main job is to either make static scheduling choices, mainly for lower-levels of hardware, or insert a dynamic chooser, for higher levels. The choices have a quality, as seen in the previous chapter, and that quality is determined by the abilities used while making them. Those abilities are in turn enabled by the information extracted.

Scheduling constraints in the application code limit static scheduling choices, which means they limit the changes to processor specifications allowed to the specializer. Those specifications in turn constrain the executable code generated, which can be viewed as a number of static scheduling choices of instruction-order, register-communication, and cache-request order.

Static scheduler assignments of tasks to locations at times can only be chosen statically with high quality for parts of the application with high static predictability, when the data has no effect and the hardware is also statically predictable. In general, static predictability doesn't exist. Un-predictability increases with the number of scheduling decisions in a row, limiting predictability to a short distance into the future of a run. This is why specialization has to inject a scheduler that attempts to optimize during the run.[3]

The information specialization gathered or encoded about the hardware affects static scheduling decisions. In addition, it sets which dynamic scheduler is injected inside the animator implementation.

Scheduling takes place in the execution model's animator, and specialization inserts that animator, extracts prediction information to help it schedule, and manipulates code to improve the decisions its scheduler is able to make.

Often, the best static scheduling choices for a given task's code cannot be decided before knowing the dynamic scheduler's task placement, or even input data. For example, a given task may be assigned to a CPU or to a GPU, which require very different task code! In this case, the toolchain specializer generates a manipulator for the animator to invoke. In essence, choosing between the two code images for the task amounts to specialization performed during the run.

[3]In the parlance of the Basic Model, specialization changes the processor specifications, in order to change the tasks available to the animator's scheduler, it changes the processor specifications for the sequential portions of code, which implicitly controls the animator's scheduler, and it injects implementations of higher-levels of the animator hierarchy, matching the implementations to the hardware. It extracts prediction abilities for the schedulers in these animator levels to use.

8.4.7 Search During Specialization

For portions of code with sufficient static predictability, it performs a search through sequences of scheduling choices, and picks a sequence that optimizes the goal measure, then embeds that sequence into the code. For portions of code with too little predictability, it injects an animator-implementation that receives the prediction-methods and uses them during the run to perform limited searches that locally-in-time maximize the goal measure.

8.5 Existing Specialization Techniques

This dissertation describes two different general specialization approaches: hardware-specific modules applied in a central server (BLISS), and a three-phase specialization approach that relies on a generic toolchain and morphable execution model (VMS).

Another popular approach is library-based specialization, where the hardware-specific library is linked during the run (Library-based). A variation on this is just-in-time compiling, which also inserts hardware-specific library implementations during the run, but can also in-line and perform limited other code transformations (SEJITs).

Traditional approaches rely on fully-static scheduling. Loop-nest oriented parallelization techniques generally fall in this category.

For VMS, the generic toolchain has to be effective for a variety of hardware architectures. It does this by only performing part of the specialization, which is extracting the information that a dynamic scheduler needs to make high quality decisions. The scheduler is linked when the application runs.

A major limitation is that the toolchain only has limited knowledge of the hardware, so it can't make task-boundary choices nor low-level static-scheduling choices.

For BLISS, potentially the full hardware is known, so the toolchain, possibly together

with a specialization expert, can specialize all the levels of hardware. It is unclear whether fully automated specialization will be able to deliver performance as high as hand-tuned code, but tools like SPIRAL [11] suggest a possible path.

Both BLISS and VMS are broad-based, placing no inherent limitations on applications nor language.

In contrast, library-based approaches are applicable for only a narrow range of applications, notably ones that flow data among a graph of standard transforms. This is a popular class of applications in scientific and signal processing domains, but less so in general.

8.6 Effect of Multiple Execution Models in the Application[4]

Source code in some languages, such as C, have library calls available that actually escape the C execution model and invoke behavior from an embedded execution model. OS calls make a good example. C does not define the behavior of OS calls! The library uses a hardware mechanism to switch between the C animator and the OS's target animator, which then makes the OS behavior go forward.

This mix of execution models is implied in most current parallel languages, which combine concurrent behavior with sequential behavior. Programs normally consist mainly of sequential code, with a sprinkling of parallel constructs to define task boundaries, and to either state or implement constraints on task overlap and order.

When one or more of the execution models have an animator that is not injected by the specializer, then the specializer is at a loss. It has no way to model the effect of invocations

[4] An execution model is the behavior and interface of the target animator, plus the collection of primitive processors available.

of the foreign execution models. This eliminates its ability to reliably align multiple levels of the hardware hierarchy, and makes predictions poorer. Specialization is only harmed by multiple execution models.

8.7 Suggestions for Future Specialization Techniques

With the preceding background in hand, what is needed to achieve high performance portability becomes clearer. This section describes some techniques motivated by the background. The frameworks of part 2 are designed to support these techniques, and a few are starting to appear in other recent work, such as CnC, Sequoia, and SEJITs.

8.7.1 Standard Information from All Applications

One interesting possibility comes from the scheduler abilities discussion in Chapter 7 on the Holistic Model. There is a fixed set of information needed to make scheduling decisions. The application is irrelevant, the needed types of information stay the same. To get high quality scheduling decisions, this information is then combined with hardware information, as discussed.[5]

The nature of the application determines how useful the information will be, and the nature of the hardware does as well. But, the categories remain fixed.

This suggests defining standard types of information and standard formats for passing it between application-analyzer and consumers. The analyzer extracts the information and packages it for consumption.

Typical consumers will be processor-hierarchy manipulation, task-boundary manipulation, and animator. Both processor-hierarchy and task-boundary manipulators will use the

[5]Sometimes the specific data in the input to a run is also required to make high quality scheduling decisions.

information in combination with hardware information to modify the code. The goal is generating a task-graph that the animator does not need to further modify in order to achieve the goal measure for a run.[6]

By consulting the Holistic Model, some suggested information categories are:

- processor specifications in the code
- mapping between processor specifications and task definitions
- task-definition format
- location of shape-control data within task-definition
- relationship between shape-control values and flow-through data
- amount of work, given a task-definition
- amount of total data, given a task-definition
- "names" of data-structure *instances*, given a task-definition
- generated manipulator that modifies shape-control values to change amount of work
- generated manipulator that modifies processor specifications within a task
- generated manipulator that modifies processor-to-task mapping

8.7.1.1 Scheduler abilities activated by the standard information

The scheduler abilities identified in the Holistic Model should all be enabled by information extracted from the application, when possible. Some applications inherently have poor predictability in one aspect or another. But the best possible should be done.

[6] Or, requires only very simple modifications, achieved via manipulators provided to the animator.

8.7.1.2 What information feeds those abilities

Specialization needs to extract task boundaries and dependencies. It also needs to generate manipulators, and generate predictive models for data-structure instance names, data size, and work amount.

8.7.1.3 Hardware limits which of that info is usable

The hardware has its own predictability. The higher the level in hardware, the poorer the communication and work-rate predictability. If the application has good predictability of patterns of requests, it is irrelevant when the hardware has poor predictability of responding to those requests.

8.7.2 Complications from Multiple Levels of Hardware

When the application priority is performance, strongly, then all levels of hardware have to be specialized to. Even the particular input data drives specialization choices.

As example hardware, take grid computing. The top-level scheduler sees a number of super-computers connected by global-area-networks. Each machine is different, and has a complex internal network.

Take one machine on the grid. It has a torus network on a backplane that connects nodes that each have their own main memory. Each node is a board having 8 chips sharing a single main memory. Each chip has 8 cores and two levels of coherent cache. The chips communicate with each other's caches and main memory via a protocol like hypertransport.

There is a distributed scheduler that receives machine-wide tasks and breaks them into pieces. It distributes the pieces to nodes, over the torus network.

Each node also has its own distributed scheduler that receives a task off the torus

network, and breaks it into pieces. The pieces are assigned to cores. The pieces all live in the shared main-memory, but are effectively distributed to caches by the coherence mechanism and hypertransport.

8.7.2.1 Higher levels have poor predictability:

The grid-wide scheduler sees communication links that have very broad distributions for their latency and bandwidth characteristics. So it has poor predictability for communication times. In addition, each machine has a complex internal network. This makes predicting the work-time of a task infeasible beyond a broad distribution. Hence, the top-level scheduler has limited work-time and communication-time predictability.

8.7.2.2 Lower levels have increasingly higher predictability:

Each level lower in the hierarchy the complexity below it decreases. Simpler hardware has increased predictability. At the lowest level, the hardware scheduler in the processor pipelines is scheduling individual instructions onto function-units. It has perfect predictability, yet just one level up, the pipeline as a whole has worse predictability because of variable cache times and reordering.

8.7.3 Strategies for Handling Multiple Hardware Levels

The hard part of getting good performance for a given application on given hardware with given input data is the interactions among multiple levels of animator. Caches tend to be the hardware layer most unpredictable, non-stationary, and sensitive to exact scheduling choices of higher levels. This tends to bubble up, reducing predictability available to higher level schedulers and making their choices fragile.

When performance is the overriding top priority, the approach to handling this sensi-

tivity, to particular scheduling choice, has been to hand-tune low-level code choices. This, of course, defeats portability, and adds considerable expense to code development.

In the future, specialization might use a number of approaches to handle the complexities of multiple levels of hardware.

8.7.3.1 Add downstream modules:

For VMS, there is a problem. The toolchain performs specialization before distributing the installable image, and is generic. It is limited in task-boundary choices and transforms of lower-level scheduling choices. Not until run time is the scheduler for top-level tasks linked in.

To alleviate this, the toolchain is split and distributed. The generic portion that extracts prediction and task information packages what down-stream modules will need. Then, during install, and possibly also at the start of a run, additional toolchain modules run. These are specific to the hardware, so can make detailed decisions. However, the down-stream won't be able to interact with a specialization expert, as the BLISS and library-based approaches can.

Both the dynamic-link library approach and the just-in-time compile approach lie in this category. A JIT is effectively a specialization module that runs intertwined with the application.

Another example of this approach is for the upstream specializer to insert into the binary a later-run specialization module. An auto-tuner might be done this way, invoked when the input data becomes known.

The distinction is that for install-time, the downstream modules are separately installed on the target hardware, and for initialization-time, are invisibly injected into the executable itself. In both cases, the specialization module runs down-stream from the main toolchain.

8.7.3.2 Scheduler overrides:

The programmer can sometimes find places in the hierarchy of task-decomposition where a hand-tuned scheduler performs better than the one inserted by the tools. In this case, the tool-inserted scheduler is overridden.

A likely place is a tight kernel. For example, the programmer may analyze specific cache effects and make instruction choices accordingly. This is packaged as an override, and the toolchain inserts that hand-tuned version of the kernel.

Other examples would be where a narrow-focus tool sometimes can generate high quality custom schedulers. Polyhedral techniques are a good example. The tool is only useful when certain criteria are met by the hardware, the application, and the data. In this case, it can generate a scheduler that will outperform the general-purpose one normally inserted by the specializer.

One mechanism would be to include a command that tells the standard scheduler the presence of the custom scheduler and its requirements. Requirements include things like owning a block of processing elements, all of which are assigned to the task instead only a single one. The standard scheduler would modify its scheduling activity to set aside the resources, then invoke the custom scheduler in the task, giving it ownership of those resources.

8.7.3.3 Manipulator for each level:

Simpler manipulators for lower-levels would be a selection of kernels. The hardware at a level up might be affected by which kernel was in use below, and its scheduler would choose a task-size in coordination, using a higher-level manipulator. And so on. The idea is to have an animator per hardware level, and manipulator at the granularity of the hardware at that level. The animators communicate in order to coordinate their manipulations.

8.7.3.4 Multi-versioned task-code:

Multi-versioning could also be viewed as a form of manipulator. Upstream specialization makes several versions of task code, each with static scheduling decisions made that optimize a particular context. The properties of input data would be a context that drives choice of task-code version.

The point is that a coherent, efficient arrangement was found that makes multiple levels of the hierarchy work well together, for example on a GPU, where data-layout affects memory-fetch efficiency, given the banking and pipelining. The data layout and corresponding code of several kernels have to be modified together. The kernel code choreographs the memory and processor.

8.7.4 Hardware Cooperating with Specialization

Communication above a certain hardware level is too costly in comparison to instruction time. Control over that communication should move from hardware over to software, allowing the animator injected by specialization to control the more costly communications.

In general, higher the consequence of poor scheduling decisions,[7] the more sophisticated the animator's scheduler should be, and the more think-time it should take. There's a point at which fixed, closed-off hardware schedulers should give way to interaction with software, and another point at which they should be replaced with programmable schedulers. Programmable can be FPGA-style, or Von Neumann style, or some combination.

The lower the level at which control is given to specializer-injected animators, the more opportunity for application to influence animation. This enables coordination between levels of

[7] One measure of consequences is the ratio of time to communicate one byte vs do one instruction on it, another is the ratio of worst-case schedule to best-case. For example, L1 cache, even if all miss, it's only about a factor of 3 slow-down, assuming 20% LdSt mix, 10 cycles to L2 and all hit in L2. However, if all L2 miss to main-memory, it's a factor of 200 to 1000 times slow-down. Hence, L1 is fine with a hardware scheduler, but L2 should be controllable by a specializer-injected animator.

hardware that aren't possible with fixed hardware schedulers, especially cache schedulers.

8.8 Topics Related to Specialization

A few interesting topics related to specialization are collected in this section.

8.8.1 More on How Caches Affect Specialization

A cache is its own collection of processors: a top-level processor transforms address and checks the tags, while inner memory processors contain tags and contents. It has its own algorithm for what addresses are present, which determines when it generates communication.

It can be modelled as an M onto N memory processor with statistics for when it generates communications. However, it is non-stationary, so the statistics are only accurate for the exact programs plus data they are gathered on, and are often extravagantly in-accurate for a different program or different data. Matrix Multiply is the standard example of repeatable poor cache behavior that consistently violates the statistical model of a cache.

This fundamental inability to model cache-generated communication accurately presents a problem for scheduling. The fundamental aim of scheduling is to choose data-to-location plus operation-to-location such that consequent communication patterns maximize some measure. This can't be done if the communication patterns have no reliable models! It has been dealt with for sequential specialization by detecting particular code-patterns that cause repeatably poor cache behavior.

But this solution was only practical for sequential programs because they have a fixed scheduling of the instructions, with a relatively fixed address pattern, which gives predictability. However, with multiple cores interacting with the same cache in patterns that are created at runtime, there is too high a complexity, involving history of past scheduling decisions and traffic

from other caches, to predict the patterns accurately. As a result, cache-generated communication is unpredictable for parallel programs on parallel hardware that has hardware caches.

All that can be done is hope that especially inefficient patterns are rare, and rely on statistics gathered in particular configurations. The issue is that the additional degrees of freedom introduced by the dynamic scheduling in parallelism make such statistics less reliable, in general, than they were for sequential programs on sequential hardware.

The mechanism causing the problem is the internal structure of the cache, which has strong non-linearities that create a large number of sharp deviations from average behavior. In particular, the matrix multiply behavior is due to the hash-function, which simply truncates bits, causing an address pattern which varies only in the truncated part, to use a sub-set of the cache's internal memory.

Use High-Entropy Hashes in Caches: The sharp deviations from the statistical model can be mitigated with a high-entropy hash that distributes addresses with greater variability [52]. The high entropy makes it less likely that a very regular program[8] will generate an address sequence that repeatedly maps onto a subset of the internal cache memory. It effectively smooths out the observed cache behavior, which makes the statistics more reliable. In turn, having a statistical model of a cache that is rarely deviated from allows dynamic schedulers in parallel systems to use simple statistical models. They don't have to worry about particular patterns causing strong deviations, so outcome from their decisions will match closely to predictions, increasing the reliability of their scheduling decisions. Reliability is more important than absolute best performance in many real-world contexts.

[8] Programs are necessarily regular because they are generated by people, who manage complexity by exploiting regularity. The very semantics of programming languages causes a high degree of regularity of address patterns. Consider writing a program that accesses matrices in a highly irregular way without using a random number generator. High entropy patterns simply happen rarely in human-generated programs. The human has to explicitly try for high entropy, and in such instances will almost never write code that repeats the irregular pattern.

8.8.2 Amdahl's Law in Specialization

Amdahl's law [1] is a statement about scheduling constraints. It was defined for speedup in a processor, where only a single level of scheduler existed. In contrast, for parallel software, multiple levels of animator can be defined, each with its own scheduler. Each individual level is subject to Amdahl's Law, but the overall hierarchy is not! This upholds Amdahl's law, but suggests it may not impose a harsh limit.

For example, in matrix multiply, one level of scheduling corresponds to one level of blocking of the matrices. If the hardware were such that very small blocks gave the best performance, and the program only had a single level of blocking, the portion of code that created all the blocks, and handed them out to processors, would require more processing than one of the blocks! This is Amdahl's law: creating the blocks and handing them out is done sequentially, and limits the speedup. However, a level of hierarchy can be added, so the top level creates processors that each create a number of blocks. The size of sequential at each level of the hierarchy is then smaller. If it's too big, add another level.

Restatement of Amdahl's Law: Amdahl's law can be restated in terms of tasks: the time of the propendent dominates if it is longer than the longest dependent. In this formulation, the propendent is the task with the inherently sequential code. Total running time is greater than or equal to propendent iteration-time, and less than propendent iteration-time plus longest dependent iteration-time.[9] Hence, total running time cannot become less than the sequential portion inside the propendent task.

The consequence is that, the hierarchy of processors implied by the code plus data must be mutable, and then assignable to tasks in a way that matches the physical patterns.

[9] Virtualization may affect the overlap, where lower-level animators inter-twine propendent iteration with dependent iterations, and overlap portions of communication. Also, the work of the propendent may be creation of the dependents, which makes each dependent see a different work-time of the propendent iteration.

The time spent in propendent tasks should be made less than or equal to time spent in any *directly* dependent task. A dominant propendent must be either, amortized by the length of path through it to the end of the run, or else overlapped by non-descendent iterations.

The relationship to sequential sections of code becomes a question of the task-graph. Can the task with the sequential portion be divided into a hierarchy? If not, can the tasks be re-formed such that the scheduler-loop iteration with the sequential code is overlapped with iterations unrelated to its descendants? If not, are there enough descendants to amortize the dominant task? Its direct descendants are all too short, but is sum of all dominant tasks a small overall percentage of total running time?[10]

The question of parallel applications then becomes: How many problems impose a strong limit to the depth and shape of the software-processor hierarchy? That hierarchy constrains the task-graph, which constrains the iteration overlap which sets total running time. In looking at this question informally, it appears so far that the most common limit is unnecessary human-imposed constraints.

8.8.3 Standards Artificially Limit Parallelism

The MPEG family of standards provides an example of human-imposed limits on processor hierarchy. It limits adjusting the task dependency graph. Looking at the relationship between standard and processor hierarchy: the standard defines one level of macro-blocks: this limits hierarchy depth. It defines the size of the macro-blocks, which limits the size and number of children. It also defines operation-order within the blocks, such as the scan-order in the deblocking filter. This adds scheduling constraints that have little impact on the qualitative result and further constrain the shape of task dependency graph and consequent iteration overlap.

Similar artificial constraints on hierarchy and task-graph can be identified in other

[10]The dominant task-times are added together and amortized by the length through non-dominant.

types of standards. Hence, the parallelism limit in these is an artifact of human choices. The problem likely could be solved in alternative ways that achieve similar goal-measures. The goal for such standards are complex, such as level of compression, quality of image, and so forth. But change should be possible without *artificially* limiting the mutability of the software-processor hierarchy.

Part II

Frameworks that Support *Collectively* Meeting the Triple Goal

In this part, the background is put to use. It conveys two frameworks to support collectively searching for a satisfactory solution to the triple challenge. Those frameworks will themselves become a central and lasting part of that solution.

Chapter 9

Overview

This chapter draws the big picture, describing what functions the frameworks should serve, how to evaluate whether they meet those, and describing how adoption of the frameworks might progress over time.

The triple challenge of productivity, portability, and adoptability for parallel software is a large, complicated problem. Independent groups are individually too small to solve it, and need infrastructure that lets them focus on just one part of the problem. The infrastructure then integrates their research with that of others.

The issues such a framework has to be coherent with are many fold and interrelated. Most important among these, it has to support a variety of languages and development processes that fit with existing ones as much as possible, increase the rate at which new parallel constructs can be tried out, support the specialization process, and allow reuse of specialization work across languages and across hardware.

The lifecycle of such a framework will be to start as a helper for research, then remain as permanent infrastructure as the research becomes adopted into the segments.

9.1 Framework as Permanent Infrastructure

The framework will transition from research helper to permanent infrastructure. One reason is that advancement will continue, so the search benefits of the framework will continue to be needed. In addition, independent entities in a segment will supply different parts of the specializers and target animators, so the integration role of the framework will continue being needed. Finally, the specialization process is intricately tied-in with the framework, making separating the two infeasible, so the framework will be needed for the specialization process.

Hence, such a framework will become a new element in software infrastructure, alongside operating systems, network protocols, and so on. Like an OS, it will become a segment-wide standard, in order for different entities to effectively work together and interact.

9.2 Functions a Framework Serves

To support the search, a framework must support:

Developing new languages The framework provides all the pieces below the language level, so the designer can focus on just the language design aspects they're interested in. At a minimum, they only need provide a plugin and library wrapper to implement a new language – perhaps a few days of effort. Ideally, the framework provides APIs and other support, such as the VMS interface, for rapidly prototyping new language ideas. This is especially valuable for exploring productivity in terms of mental model the programmer creates. New languages can be quickly tried to zero in on what matches their natural internal mental model the closest.

Specialization research The framework provides an end-to-end solution, so the researcher can just replace a poorly-functioning piece with their better performing idea, or can add

a special-case tool that works for a small subset of patterns in existing programs. The framework should minimize any effort that is not directly the idea of interest.

For BLISS, this means cloning a working system and replacing some existing piece with the research version. The applications are there, the languages, and so forth. None of that works needs to be done before trying the research idea.

Execution Model/runtime research The framework provides a number of languages that have runtime implementations, and have a number of applications ready to run. This allows and apples-to-apples comparison between different techniques.

Creating new parallel development tools The framework provides an adoption path and an assortment of applications in languages already running. New debugging tools that require modifications to the execution model or runtime are no problem – everything is within the framework. Verification, testing, documenting, these can be explored in terms of an already-working system. Process, workflow, modelling the problem – the framework is the standard place that such tools will be distributed along with whatever languages they were designed to work with.

Integrating all the above, when performed independently The one-stop convenience and integration provides logistical and adoption benefits. All the functions are served, and adoption has been studied and the framework aligned with it, so any piece of the puzzle just has to fit the framework, and it gets the adoption ride-along included.

9.2.1 Framework as Specialization Infrastructure

The framework should function as infrastructure for the specialization process:

Specialize after development in segments such as embedded and shrink-wrap, specialization steps must take place after the code is behaviorally complete. Development would slow

too much if specialization to all targets was performed inside the development iterations. Hence, infrastructure will be valuable for post-development specialization, especially if it can automate portions of the process, and establish a standard for testing and verification tools as well as specialization tools.

Combine contributions in standard way Low-level hardware portions of specialization are combined with language-level portions. The combination is applied to the application code. Having infrastructure for this reduces duplication of effort, by establishing a standard. Without a standard, hardware manufacturers are currently duplicating the low-level specialization portion for each different language. This increases the development cost within the embedded segment, where the pain of it is already high and accelerating.

Reuse effort The specialization is different for each hardware, but effort should be reused. Such reuse is more practical and effective when standard infrastructure defines interfaces and behavior by which elements interact.

Specifically, there is a potential explosion of languages times hardware, each combination needing its own specialization tools. A standard that improves reuse will keep this manageable.

Language integrates with specialization Specialization needs language structures to support it, so need infrastructure that holds both, the language and specializer, to make it natural for them to integrate.

The application has to be analyzed. This discovers: what are the tasks, the constraints on them, their predictability properties, and how to split tasks up or fuse them together. The languages need a standard target format to package this into, so that it can be reused across machines. The infrastructure tells the language designer what needs to be extracted, which clues them in to features to put into the language.

9.3 How to Evaluate the Quality of a Framework

A framework has to be evaluated on how well it serves its purpose. This is two-fold: 1) to act as research accelerator and integrator for independent research groups, and 2) to act as permanent infrastructure in one or more software segments.

As a research tool, it must be demonstrated 1) that research productivity is increased in each area needed to meet the triple challenge, 2) that independent research is naturally integrated by the framework, and 3) that the framework aligns research with the three goals, as a byproduct of the research being performed within the framework.

As infrastructure for software, it must be demonstrated 1) that specialization is supported, 2) that logistic issues and security issues can be reasonably handled by the framework (for long term success), 3) that the framework is aligned with development and distribution practices (to address a barrier to adoption), 4) that it is free from other potential adoption barriers (such as performance barriers), and 5) that the infrastructure supports the array of applications in a targeted segment.

Separately, to become infrastructure in a software segment, it has to be adopted. Hence, a case should be made for a reasonable path to adoption. This should include an analysis of early adoptors, and steps in the growth of popularity. At each step, the push, pull, barrier and uncertainties should be discussed.

9.4 Process of Adoption

Adding such a framework, as a segment-wide standard appears to be disruptive, however, as shown in the following chapters, this can be minimized by careful design of the framework. Chapter 4 gave insight into how adoption of such a framework would progress, and what the major factors affecting adoption are.

Before anyone will adopt, a seed framework must exist to be adopted! Research funding will drive the seed framework. It will have a few languages, a few applications written in those, and a few specializers to target hardware platforms. This is the starting point.

Given the seed, other researchers will adopt the framework for their targeted interests. Each will independently improve a piece of the infrastructure/tools. This is where the framework's ability to integrate will prove crucial. Eventually, the research contributions will provide enough functionality and performance that non-researchers will start looking at the framework.

The first non-research group to adopt will be individual developers. The elements for adoption must be satisfied. The pull for them comes from promise of technical benefits, the "that's neat!" factor, and buzz about it, "have you heard about!". The barrier to adoption is mainly learning curve, and effort to get it up and running.

The rate of spread depends on buzz generation, which spreads the word, initiating trying it out. The better the technical qualities, and the better the tutorial and other documentation, and the easier to get started, the faster the buzz will grow. An entity actively working to create buzz can also increase rate of spread.

Once the buzz has grown, and individuals have looked at it, teams in small companies will start using it. The push for them is the difficulty of using other parallel coding approaches. The pull is potential advantage over competitors, and desire for a more pleasant coding experience. The barrier will be the lack of mature development tools and lack of specializers and their poor quality for particular hardware they want to target. The uncertainty will be in whether the experience really is better, whether it really is the wave of the future, and whether specializers for desired target hardware will materialize. These uncertainties will prevent large entities from jumping in early, leaving small entities and skunk-work projects to be the drivers of change.

Alternatively, one large entity might be an early adopter. This will only happen if the decision makers feel they have control over all uncertainties. In this case, the entity would develop

a language, development methodology, and tools to support it, all in addition to advancing the framework and creating specializers for the hardware they want to target.

This scenario will only happen when a very large entity has severe pain and no viable solution exists. The push is the pain, the pull is competitive advantage, and the uncertainty is reduced by controlling all aspects of development themselves. The barrier is the high cost of developing all the pieces and training people in the new process. This can be reduced by proper design, to fit with existing practices as much as possible.

The barrier may not be a negative. Such a barrier is sometimes seen as an opportunity by ambitious decision makers. They see the chance to be the first to solve a pain that all competitors also have, and the size of the barrier represents an impediment to the competition. The fact of the solution being available to them exclusively is seen as a weapon in their hands.

This scenario depends on connecting with a risk-tolerant decision maker in such an entity. The entity will progress the work in stages, in order to iteratively reduce their risk. Each stage grows successively larger in budget, and successively reduces uncertainty.

Chapter 10

BLISS: Bidirectional Library Interface + Specialization Server

"Ying and Yang Diagram"

The first framework is based on three things: make the scheduler a separate library called by the application, make the application appear to be a library that the scheduler calls, and make specialization be done in a server, where all the applications are stored.

The advantages are that it fits with existing development workflows, simply adding an extra step on the end. It doesn't force the programmer to alter the tools they use nor the base language they use. This has big advantages in the Enterprise segment because processes tend to be dominant there, and any forced change to their development workflow would create a barrier to adoption. Likewise, the shrink-wrap segment often has in-house custom development processes, so forced changes to development process would create a barrier to adoption there as well.

A second advantage is that the server-centric approach allows complete flexibility for the toolchain. The specialization server is designed to take toolchains as black-boxes, including

scripts that then call tools inside the toolchain. This enables sophisticated toolchains that have collections of tools from multiple independent sources, such as compilers, source-to-source transforms, and so on. This approach applies the minimum possible constraints on the toolchains that perform the specialization.

A caveat is in reuse. The high degree of freedom comes without standards that would increase the reuse of portions of the toolchain. But such standards are not prevented, they simply aren't imposed by the first framework implementation. It is anticipated that once experience is gained, common patterns within toolchains will be recognized and a standard generated that formalizes these patterns.

For example, most toolchains will likely have a portion that identifies the parallel tasks, fuses or splits their code, and generates manipulators for the animator (runtime) to modify the shape-control data of each task it schedules. It may be possible to define standard interfaces for each of these specialization functions, which would in turn allow reusing an implementation across toolchains.

The specialization server collects application sources, then, to each, applies multiple toolchains. Each toolchain specializes to a different hardware target. Each specialization starts with the same source and ends with a different installation-bundle, for one target.

The code that performs specialization is collected from many 3rd party sources. These specialization modules are plugged in to the server's automated system.

Hence, in BLISS, there are two distributions in play: collection of specialization modules, with distribution to specialization servers, and then collection of applications onto specialization servers, with distribution to end-users.

What has been accomplished: Schedulers for several hardware platforms have been implemented, along with infrastructure that performs some automation. In particular, the infras-

tructure automatically specializes any instrumented Java application to run on a shared memory multicore machine, and produces a runnable jar file specialized to the target hardware's number of threads.

How to measure goodness of BLISS Framework: In order to be convincing, what must be demonstrated is: that an array of applications can written to a BLISS style interface; that a number of schedulers for different hardware can be supplied; and that it is easy to link the two.

What must be described is: the interface; the framework implementation; and the process of plugging into it applications and schedulers. It must have sufficient detail to asses the level of convenience of using the framework.

Breadth of Applications: The bidirectional interface approach can support many forms of parallelism, including stream (pipeline) parallelism; functional-block (component) parallelism; and parallel-library-implementation parallelism. It can also support general communication between work units; atomic updates of shared data structures; parallel distributed I/O; nested levels of parallelism; and runtime prediction of computation and communication costs.

Real-world issues – API changes: Compatibility between applications and server is important. Old applications should work with servers implementing newer versions of the standard, and New applications should work with servers implementing older versions of the standard.

A process is envisioned for BLISS to retain backwards as well as forwards compatibility. The key is to use source transform tools designed for API changes, such as DAMit [2]. An old application will work with a newer hardware-specific library, while an old library works with a new application. This insulates application source code from changes in the interface.

New applications implement additions to the interface as desired, but those are not guaranteed to work with older hardware implementations. This will never cause incorrect re-

sults, but it may cause potential performance to be lost. Older schedulers will ignore interface additions, and new interfaces will only add to existing, never change. This scheme makes it easy to get started on a new application: the developer is free to implement only a minimal sub-set of the interface. As more performance is desired, extensions are implemented as needed.

Accelerating search: This scheme supports a rapid search for the best interface between application code and hardware-specific code. Both sides are protected from changes in the interface, letting the search for the best one progress freely.

Chapter organization: Section 10.5 describes the interface defined for data parallelism. Section 10.6 describes the three sample applications and how each was instrumented with the pattern. Section 10.7 describes the automated infrastructure. Section 10.8 describes the details of the specializers and the run-time schedulers in them. Section 11.6 gives brief execution-time results demonstrating successful use of the framework for the three hardware platforms and the three applications. Section 11.7 concludes the paper.

10.1 What BLISS is

The framework is called Bidirectional Library Interface + Specialization Server, or BLISS. The "bidirectional library interface" part refers to the fact that the scheduler is a library that the application calls, while the application is a library that the scheduler calls, and the "+ Specialization Server" refers to automated specialization that takes place on a central server.

Languages that fit within the BLISS framework all have one thing in common – they include constructs that, in effect, provide the implementation of a library interface. These are used by the scheduler during the run to create tasks, choose their size appropriate to the hardware, decide the best place to run the tasks, and predict the communication caused by the

task assignment.

It chooses post-development-pre-distribution as the time of specialization, and single-distributed-server as the place for its first, and main, specialization step.

However, the server-based specialization step is free to insert additional down-stream specialization steps into the installation package. Both (install-time, on-the-client), and (run-time, on-the-client) specializers may be injected. BLISS asserts as few constraints as possible, while asserting all the constraints needed to enable the portability goal, which requires careful balancing of competing goals.

BLISS can be thought of as encompassing three things: a methodology framework; a set of libraries or development tools; and a server that performs the (first) specialization step. The programmer follows the methodology, inserting calls to the BLISS library and providing implementations of the reverse-library calls. They test on their local development hardware in a normal development cycle using their traditional sequential development tools.

BLISS takes care of specializing the code to the various hardware platforms. It isn't a panacea and can't do the impossible – it's just a platform that the real, difficult, work of creating specializers is plugged into. However, having such a platform will be a boon to the portability challenge by bringing together the work of diverse groups into a single place. This sort of aggregation has real-world benefits and, as a result, tends to establish standards.

Benefits of BLISS: Benefits include a critical mass of specializers for the application-writers to write to, and conversely a critical mass of applications for specializer writers to use as test cases and benchmarks for comparison – Ying and Yang.

Each high level pattern of parallelism can have its own biLib, centered around a scheduler for that kind of parallelism. The first biLib is for data parallelism (Sec 10.5), while future biLibs will be for stream parallelism, function-unit (component) parallelism, and Library Par-

allelism (a library of common communication-patterns or constraint-patterns that have hand-tuned parallel implementations).

For application development, the framework should be made as "turn-the-key" as possible, allowing an application developer to write an application then "press a button" to get an executable for their target machine. The proof-of-concept implemented for this dissertation demonstrates one path to accomplish this.

In practice, especially early on, the specialization step will likely involve programmer interaction. A new class of tool will emerge, which allows interactive modifications to the code, while automatically running regression tests that cover the affect areas of code. This will ensure that behavior has not been changed by the code restructuring performed by the programmer. A class of programmers will likely arise, who become experts in specializing to particular types of hardware.

Meanwhile, for researchers who wish to focus on one aspect of the problem, such as Code Structure Transforms (CSTs) or schedulers for specific hardware, the proof-of-concept implementation shows how to make it easy to "plug in" such research.

How BLISS supports collective solution: BLISS supports the collective solving of the triple challenge by enabling independent researches to replicate a working system, focus on one part in isolation, then contribute their work back. The decoupling provided by the bidirectional library pattern enables using a specialization server. The specialization server accumulates applications as well as specialization tools and the embedded, hardware-specific, schedulers and runtimes. That accumulation allows replicating a complete working system, which in turn enables a group to modify only the portion they wish to do research on.

Hence, a group can do focused, isolated research, but because it's done on a replicated working system, and because of the clean interfaces within that system, their isolated research

is compatible with that of others working on other parts, within their own replicated systems. When done, each contributes back to the main branch, so the independent work gets collected and made available for subsequent research and general use. The integration of the independent work is automatic, by virtue of starting with the same system then contributing back, and by the system being decomposed such that the work on schedulers is independent from the work on languages design, and so on.

10.2 Programming Patterns

The following discussion talks about bidirectional library patterns. However, custom languages are implicitly included, if their constructs can be transformed into such a pattern. For example, the specialization toolchain is free to first perform a source-to-source transform that turns custom syntax into an equivalent bidirectional library pattern embedded into C.

The first programming pattern to be implemented is one for data-parallelism, called Divider Kernel Undivider, or DKU. It explicitly demarcates the code of parallel tasks, and defines a manipulator that modifies the shape-control data, which changes the amount of work in a task (refer back to Chapter 6 for more on shape-control data and tasks). The execution-model's animator (the runtime) calls the manipulator to tune the work to the characteristics of the hardware.

This pattern only covers problems that are data-structure centric, like matrix-multiply or mesh-based simulations. Also, the individual tasks cannot communicate with each other. An extension adds arbitrary communication between tasks. The extension has been designed but implementation remains.

A planned second family of patterns is oriented around timelines, and called Frame of Reference, or FoR. It will cover problems with more control aspects and communication

aspects. An example problem is a signal processing system with various transforms that data flows through. Another good example is an NP complete problem that uses heuristics to jump around in the search space.

With this family of patterns, collections of data and tasks are explicitly assigned to particular program time-lines. All tasks within a time-line are performed with standard sequential ordering. Explicit manipulators can split or fuse time-lines and move data or tasks between them. A task in a timeline can receive data or send data to other time-lines, and *define constraints on its delivery.* In addition, explicit constraints can be stated on initiation of tasks in one time-line relative to initiation of tasks in another time-line. This is intended for declaring partial ordering of tasks in a natural way, while leaving the maximum freedom in scheduling.

Such a pattern provides both time-ordering concepts as well as data-relationship concepts, and should fit naturally to problems with more control orientation. After all "control orientation" means "constraints on the ordering of tasks", and the pattern includes explicit constraints on the ordering of tasks relative to each other.

DKU has been defined and implemented, and several applications written in terms of this pattern. Basic FoR has also been defined, but remains to be implemented.

DKU has also been embedded into the design of a language for the VMS system, as described in the next chapter. The language, called WorkTable, has been partially implemented. Not only does it include the DKU pattern, but also has specialization support features that are outside the scope of a bidirectional library approach. In fact, many of the limitations of the biLib approach are relieved by the VMS approach.

The Importance of Separating Out the Scheduler: As seen in the model of parallel computation, scheduling is at the center of parallel execution. In keeping with this, BLISS makes the scheduler a first-class entity that is directly communicated with by application code.

A complete dialog is possible, for example while negotiating on the best way to divide a large work-unit (task) into smaller ones. The task may not be dividable into regular-sized pieces, or the requested size or number of pieces may not be possible. Hence, finding the optimal division that maps in the best way onto the hardware, given current hardware state, involves both complexities of the application and complexities of the hardware. Iteration between the application-part and the hardware-part is necessary, hence the dialog between the two.

Such a dialog is only possible when the scheduler is made an explicit entity. The effect is isolation of scheduling from application, which is necessary for specialization. The benefit of a dialog between application and scheduler for task division is high performance on complex code across a very wide range of hardware. A seen in the chapter on specialization, the choice of task size is one of the three most important aspects for high performance in mapping an application onto hardware.

Limited ability to resize tasks accounts for the limited range of hardware that other attempts at performance portability have suffered from. It is the most important factor – the greater the ability to resize tasks, the wider the range of hardware high performance can be achieved on. Only resizing tasks is not enough – overlap of communication with computation is also needed, as is refactoring of the task code for truly wide portability. Highly constrained hardware, which has many strong non-linearities in its communication patterns, requires extensive refactoring of the communication patterns inside a single task, in order to get a good mapping of task to hardware.

Recall that scheduling is defined to consist of: choosing work-units, choosing resources to perform each work-unit, and choosing the timing of starting the work. A work unit is a tuple of a code-snippet, bookkeeping data, and work data. The code snippet is controlled by the bookkeeping data, causing the combination to transform the work-data. Hence bookkeeping data consists of things like iteration-space bounds, while work-data is a direct ancestor of the

computation output.

10.3 Bidirectional Library Interface

Fig 10.1 shows how a bidirectional library works. On the top, application-code is represented by boxes, and the arrows coming out represent calls to the interface. Arrows coming in indicate that the application-code implements one of the interface-functions. On the bottom is the hardware-specific code. Again, arrows in indicate that the hardware-specific code implements one of the interface functions, and arrows out represent calls to interface functions. The application code, and hardware-specific code, only interact via the library calls.

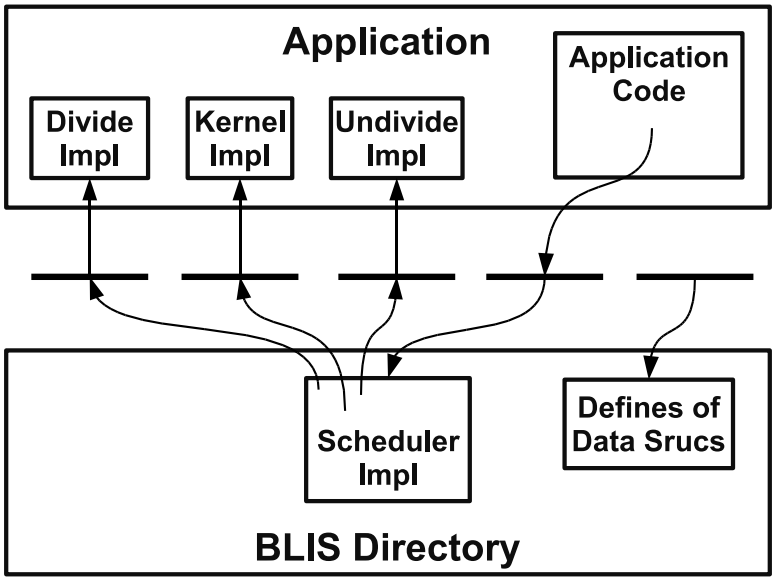

Figure 10.1: How a bidirectional library works. The application, above, implements app-lib functions, while the hardware directory, below, implements HW-lib functions. The thick lines in the middle represent the interface itself, and the curvy arrows represent "calling" the interface. To run an application on a target, the scheduler implementation for the target hardware is linked to the application.

By limiting interaction to just library calls, either side can be replaced with a different implementation of the library calls and the combination still compiles. BLISS collects all the

hardware-specific code into a directory that is placed into the source-tree of the application. This directory appears in a standard place and is called “Bliss”.

10.4 Development work-flow with BLISS

Fig 10.2 shows the work-flow of the BLISS framework. As an application passes through the flow, it first is written, compiled, and debugged in a sequential environment. The application developer uses the hardware-specific library for their development machine, which resides inside the BLISS directory, as seen in the figure. This directory contains implementations of all the hardware function calls, such as the scheduler call, as wells as definitions of standard data structures.

When an application is complete, its source code is sent to the Specialization Server [4], where it is first saved for future use, then sent to each specializer in the server. A specializer is custom to a single hardware platform, and produces an executable, or set of executables, that run native on that platform.

The executables produced by the specializers are later retrieved by the hardware on which they run. To get an executable, the client hardware runs a BLISS client that connects to the Specialization Server and requests the application. The server automatically sends the correct executable for the hardware, and the client makes that executable available to be run.

When using BLISS to perform research, a team can clone the Specialization Server and modify their local copy. This gives them a fast, responsive development cycle and supplies a ready, reusable pool of applications, which were copied-over inside their clone. The applications may be modified locally, to explore alternative interface features. When the team publishes, they can propose that their modifications be adopted into the central Specialization Server.

This streamlines the process of collaborative sharing of work, collecting the best, in

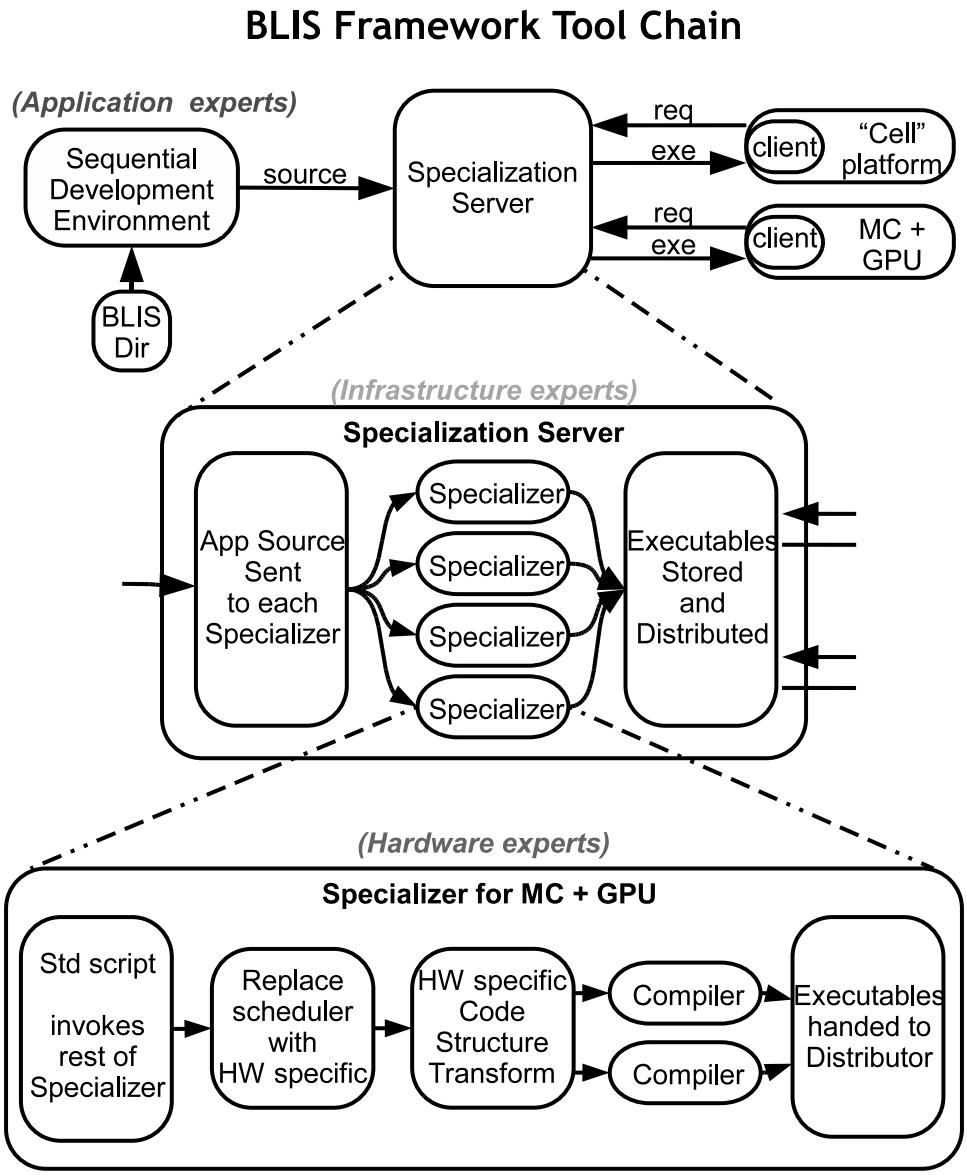

Figure 10.2: The tool chain of our sample embodiment of the BLIS framework, with breakouts showing what is inside certain elements. It is designed to encourage general researchers to write applications to the BLIS standard, and to make focused research on one aspect of the triple challenge convenient. ("MC + GPU" stands for multi-core plus GPU)

the manner of open source cooperative development.

Such a process will increase the experience of researchers with application developers, who use the BLISS interface. This will drive the interface towards the most convenient form

for productivity. Productivity researchers will also have the hardware-specific code available for modification to their needs. They can propose new forms of the BLISS interface, and validate their productivity value against the collective experiences of application developers across domains. Such interface changes will progress more rapidly than the evolution of new languages.

The BLISS interface is driven by development of the specializers. Over time the interface will explore the application-to-hardware boundary, collecting the most essential "knobs" that schedulers require, and discovering the most essential information the schedulers need from the applications.

10.5 The DKU Interface for Data Parallelism

"DKU" is the name of the biLib (bidirectional library) for portable data-parallelism. It stands for "D"ivider, "K"ernel, "U"ndivider, which are the main library functions that the application implements. The DKU-scheduler uses them to divide work into smaller pieces, execute the kernel on each piece, then collect the individual results into the larger result.

The DKU pattern's memory model is shared memory, however, this restriction is lifted by the use of the "bundling quad" interface extension described in Sec 10.5.5. A bundling quad makes the application executable on distributed memory architectures.

When the application's control flow reaches the data parallel section, it calls the DKU scheduler, passing it the data to be operated on. The scheduler has a choice of whether to exploit the parallelism, or simply pass the data to a serial kernel.

Normally, the DKU pattern replaces a loop nest. When DKUized, the loop nest is wrapped and becomes the Kernel, while the data that would go to the loop nest is handed instead to the DKU-scheduler. Hence, the serial kernel is the original sequential loop nest. It

takes the data in the application's native form, and processes it sequentially, avoiding overhead.

Figure 10.3 shows control-flow when calling the DKU scheduler. On the left, the scheduler decides the data is too small to be profitable and calls the SerialKernel. When the SerialKernel returns, so does the scheduler; the results are communicated by side-effect

On the right, the scheduler decides it is profitable to process the data in parallel and hands it to the first of the DKU functions, the Dividable Piece Maker, which packages the data into the DKU pattern's standard data structure. The Dividable Piece Maker is necessary for two reasons: being a set of library-functions, the D, K, and U must have a standard interface, so the data has to be packaged into a standard data structure; and there may be dependencies within the data that must be accommodated, so the `DividablePieceMaker` encodes those dependencies.

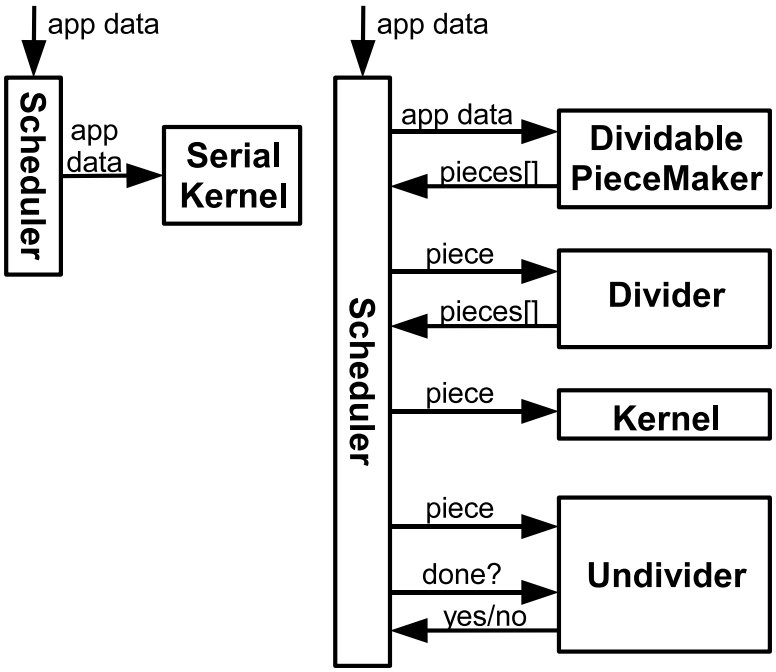

Figure 10.3: The call structure of the DKU interface. On the left is the case when the scheduler chooses not to exploit the parallelism. On the right is the case when the scheduler does. Note that the Dividable Piece Maker turns application-format data into DKU pieces. It is the bridge from application to DKU pattern.

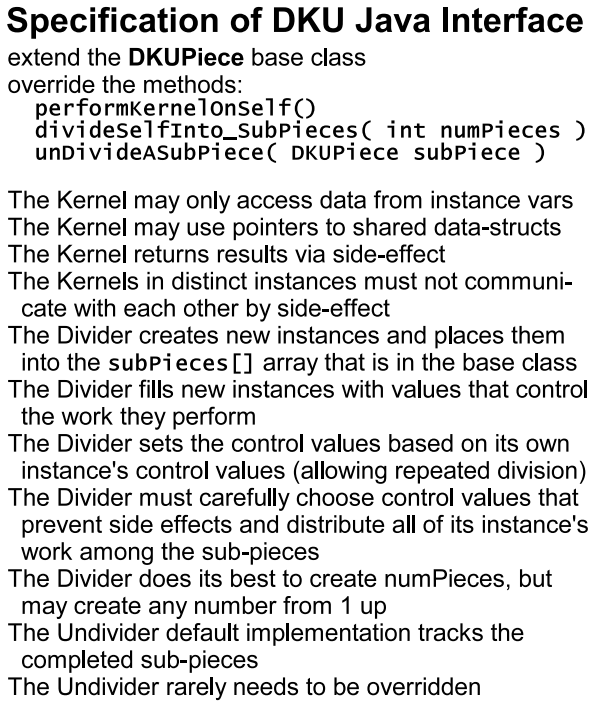

Figure 10.4: Above is the minimum set of rules to follow to instantiate a valid DKU instance in a Java application (the Kernel is the implementation of the `performKernelOnSelf()` method, the Divider is `divideSelfInto_SubPieces`, and so on).

The Dividable Piece Maker accepts application specific data and returns the largest freely-dividable work-units (pieces of work) that it can, while still respecting the dependencies. The work-units returned from the Dividable Piece Maker are scheduled in-order, which guarantees that all dependencies in the data are respected.

For each, the scheduler decides how many work-units to sub-divide it into, based on a hardware-specific algorithm, and tells the Divider to perform the division. The scheduler then hands each sub-piece to one of the instances of the Kernel, all of which run in parallel. When a Kernel finishes, the scheduler passes the sub-piece along to the Undivider. When the Undivider indicates that all the sub-pieces are done, the scheduler moves on to the next root piece, until all are complete, then the scheduler returns.

Note that the scheduler is a library function, called by the application. Meanwhile

the D, K, U, DividablePieceMaker, and SerialKernel are also library functions, called by the scheduler. The scheduler is a black box to the application, while the application is a black box to the scheduler. This generic-in-both-directions is what allows connecting all combinations of applications with scheduler-implementations. The fact that all interactions are through library calls is what ensures that compilation is error free for all combinations.

10.5.1 The Divider

The D is perhaps the most interesting part of the DKU interface. This is where the strategy of how to parallelize a section of the application is mainly implemented. Often, the only thing divided is the iteration space; a nest of FOR loops is identified and becomes the Kernel; then the nest is modified so that the iteration variables' start and end values come from a DKU-piece data structure.

As a result, the process of dividing is reduced to placing start and end values into DKU-pieces; each piece gets a different portion of the original iteration range. When a K receives such a DKU-piece, it sets the start and end values in each of its loops from the piece, then performs the loops. Each piece has a distinct set of start and end values, so each K performs a distinct portion of the work. The application programmer ensures the pieces are independent. The independence restriction will be lifted in the next release of the DKU biLib, which will include inter-piece communication.

This basic technique of identifying loop nests, then dividing the loop-bounds among DKU pieces was used in both dense matrix multiply and H264 deblocking. However, Hamiltonian Path was included in the application set to demonstrate more creative opportunities for using the DKU biLib. There, the divider turns the control flow into data, then divides that "control" data. Section 10.6.3 gives details of how this was done.

An important point is that the divider is not required to produce the number of pieces

the scheduler asks for. It simply does its best. The scheduler always checks how many pieces it actually got back and proceeds accordingly.

10.5.2 The Kernel

The K is pure application code. It is normally a loop nest that has been slightly modified such that the ranges of its iterations come from values taken from a DKU piece structure. During the DKU-ization process, the Kernel code normally has only minimal changes from the pre-DKU code. Often only a wrapper is required.

The rules for a Kernel are straightforward: all variables must be local to the kernel, all data touched is reached via the DKU piece that is passed to the kernel (pointers are allowed), and all results are reachable from the DKU piece. Isolating the kernel from the environment in this way allows it to be run in remote memory spaces without modification.

Kernels are allowed to work on shared data-structures, and to have loop-carried dependencies. The complications are handled by the Dividable Piece Maker, described in section 10.5.4, which is tasked with identifying independent pieces within the data/iteration-space. This flexibility expands the number of applications that can successfully use the DKU interface.

In C, the Kernel has the additional restriction of using only data-types defined in the BLISS-supplied headers for DKU. “int” becomes “int32” and so forth, to ensure data sizes are the same (except for pointers) in both local and remote memories. Data-structures used in the Kernel cannot have align statements in them (these will be supplied by the specializer if they are needed for the target hardware). Finally, the bundling quad (Sec 10.5.5) must handle changes in data-structure sizes that are due to differences in pointer sizes.

10.5.3 The Undivider

The U is usually the simplest of the three. In many cases, it simply acts as a barrier, counting the completed sub-pieces until all have been accounted for. However, sophisticated specializers that perform source-to-source transforms are free to analyze the application code and substitute more relaxed synchronization. The DKU interface identifies for such a specializer where the barrier is located in the code (the U), and where the dependencies among root pieces are encoded (in the Dividable Piece Maker).

10.5.4 The Dividable Piece Maker

The Dividable Piece Maker packages application-specific data structures into DKUPiece standard data structures and enforces dependencies within the data. For example, in H264 deblocking, a macro block depends on having its neighbor above and neighbor to the left being already finished. The Dividable Piece Maker encodes these dependencies. It is handed the original data structure, and returns maximally-sized independent DKUPiece structures that can be freely sub-divided. Each piece it hands back must be processed in sequence.

The Dividable Piece Maker written for H264 deblocking is given a frame of data and slices it into diagonals of macro-blocks, handing back each diagonal as a single piece. Each diagonal can then be freely sub-divided, with its macro-blocks distributed among the sub-pieces.

10.5.5 Extension for Distributed Memory: The Bundling Quad

The bundling quad is an extension to the basic DKU interface that enables specialization to distributed memory machines. As the name implies, it consists of four functions: bundleInputs, unbundleInputs, bundleResults, and unbundleResults. Their effect is to make remote execution look as though it happened locally.

Each bundling function can be thought of as a "port" for sending between local memory and remote memory. They are only invoked by schedulers on distributed memory machines, so impose no overhead on shared memory.

BundleInputs is run locally, gathers all data that will be touched by the Kernel, and returns a pointer to the bundle. The scheduler sends this bundle, via its specializer-inserted, hardware-specific, communication infrastructure, to a remote scheduler or to a remote communication stub.

In the remote memory the remote scheduler calls unbundleInputs, then the Kernel, then bundleResults. UnbundleInputs unpacks the data and returns a DKUPiece. The DKUPiece is handed to the Kernel, which produces results that are reachable from the DKUPiece as per the standard for Kernels. Then bundleResults is handed the DKUPiece. It packs all result data into an array (or object) and returns a pointer. The remote scheduler then sends the result bundle back.

The specializer-inserted communication infrastructure that runs in local memory then receives the bundled result data. The local scheduler hands it to unbundleResults which modifies the local copy of the DKUPiece, making local memory look exactly the same as if the Kernel had run locally.

This set of functions cleanly hide remote execution from both the local application and the Kernel. The application is free to use global variables, shared data structures, and so forth. In practice it has been surprisingly easy to implement the bundling quad.

10.6 The Applications

We have chosen a set of three applications to demonstrate using the DKU interface. Two of these are implemented in Java, and are more "benchmark" code, while the third is

taken from a real application written in C. The benchmarks are dense matrix multiply and a Hamiltonian Path solver (an NP-Complete problem). The real application is H264 deblocking, which was previously optimized by hand for high performance on serial processors.

10.6.1 Deblocking Filter for H264

This is the most interesting, from a real-world perspective, as it was taken from an active project [13] and optimized for high speed on serial hardware. As such, it breaks many abstractions natural to the application. For example, the "macro block" is a natural unit of data. In an object oriented program, one would include all the data for a macro block in a single object. However, in the C code, the data is flattened onto linear arrays then accessed by calculating addresses within the arrays (the arrays are static variables).

For many parallel programming paradigms, the global, shared, flattened, arrays and address arithmetic would be problematic, due to side-effects, incompatibility of single shared arrays with the language's parallel model, and so forth. For DKU, it is handled via the Dividable Piece Maker and the bundling quad, without modifying the core of the original application code, and without undue effort. The details of how this was done are too involved for the space available; the best way to see them is via the code on the website [40].

The dividable piece maker we implemented divides a frame, consisting of macro blocks, into diagonals, making each diagonal a separate root DKUPiece. Each macro block requires its neighbor above and neighbor to the left to be completed before it is calculated, so all the macro blocks on a 45 degree diagonal are independent from each other and all free to calculate once the preceding diagonal is complete. The divider simply assigns the macro blocks from a diagonal to sub-pieces.

10.6.2 Dense Matrix Multiply

Nothing special was done in the sample implementation of matrix multiply we supply. The divider slices the two input matrices into "strips". A single DKUPiece contains a pair of matrices to be multiplied. When such a pair is divided, its left matrix is sliced horizontally into clusters of rows, while the right matrix is sliced vertically into clusters of columns. Each combination of clusters is made into a separate sub-piece. This has the side-effect of discretizing the number of sub-pieces the divider is able to make. Note, however, that the division doesn't move any data in the matrices around. Rather, it calculates start and end rows for the left matrix, and start and end columns for the right.

10.6.3 Hamiltonian Path

In the Hamiltonian Path problem, one finds a path that visits every node of a graph exactly once, or else determines that no such path exists. The standard approach is to use back-tracking, which, when one examines the control flow, performs a search that has the shape of a tree. We turned the back-tracking algorithm into an iterative algorithm that represents the current point in the search as a partial-path. There is a one-to-one correspondence between each possible partial-path and a node in the search tree.

The property we exploit is that in a tree, given two nodes neither of which is an ancestor of the other, those two nodes have no descendants in common. So, if search proceeds independently from those two nodes, the two searches will remain independent and will duplicate no work.

The divider finds a set of such nodes by traversing the search tree breadth-first. It accumulates the nodes it visits, removing a node when it explores its children. Meanwhile the Kernel is written to search depth-first to increase the chances of early termination. The

undivider, in essence, performs a logical OR of all the answers. If they're all null, the total answer is null. Otherwise, it takes the first path returned to it as the final answer.

The search tree is typically exceptionally unbalanced. To handle this, we introduced a "re-divide" interface extension. The scheduler signals a running Kernel to stop for re-division, then hands the partially-completed piece to the ReDivider. For Hamiltonian Path, we exploit the re-entrant nature of the divider, and implement the ReDivider as a few "fix ups" that handle the differences between depth-first and breadth-first search.

In the case that one of the sub-trees finds a path quickly, we have also added an extension to the DKU interface that allows the undivider to tell the scheduler to early-terminate the un-finished DKU-pieces.

10.7 Specialization Infrastructure

In general, specialization can be performed in many different ways: by hand, by a build-script, or by some more general infrastructure. To support research, we have chosen to try to minimize the effort required by a researcher who wants to focus on one step of specialization such as Code Structure Transforms or specific scheduling algorithms. Hence our sample BLISS implementation has sample code of infrastructure. It has a BLISS directory with serial implementations that is supplied to the application developer, a sample Specialization Server, and a sample BLISS Client.

10.7.1 Application Development

Figure 10.2 shows that during development, the application source includes an inserted BLISS directory that is supplied as part of our framework. The BLISS directory has a DKU sub-directory that has all standard data-structure definitions and a sequential implementation

of the DKU-scheduler. By being serial, the sequential-scheduler allows development to proceed in a serial environment, such as the application developer's favorite development environment.

When the D, K, U, and DividablePieceMaker are being debugged, an alternate implementation of the scheduler is swapped in, which calls the D, K, U, and DividablePieceMaker in a serial fashion. This pattern of swapping in different scheduler implementations leaves room for more sophisticated debugging machinery. For example, a serial scheduler could stimulate timing bugs by performing repeatable inter-leaving and simulating distributed memory.

Later, during specialization, the application will be linked, in turn, to one or more hardware-specific scheduler implementations. In our framework implementation, the hardware specific scheduler's source is packaged inside a specializer module.

The use of a BLISS directory during app-development provides additional benefits. As mentioned, our BLISS implementation allows given application code to remain compatible with both future and past schedulers as the interface is extended, and modified. The use of a BLISS directory enables this. When an extension is accepted into the BLISS interface, then a new BLISS directory containing serial implementations of all the interfaces, including the new ones, is made available for developing applications, and for developing schedulers. Meanwhile, the BLISS directories inside each application and inside each specializer already in the central Specialization Server are merged with the new directory, such that "do nothing" versions of missing interface implementations are added. Hence, all interfaces "work".

The caveat is that schedulers written to new versions of the interface will not necessarily do anything interesting with applications written to old versions and vice versa. This scheme only ensures that old apps still work the same with old schedulers and new works with new, and no compilation errors happen when new is mixed with old.

10.7.2 Specialization Server Implementation

The middle of Figure 10.2 shows that the server in the proof-of-concept implementation sends the source to each specializer module. A specializer is self-contained and is invoked by a script placed in a designated directory. The server simply calls all scripts in that directory. Plugging in a new specializer module consists of placing the module's entry-script into that common directory.

The entry script is handed a copy of the application code-base. From there it invokes other scripts in the specialization module. The first script typically deletes the DKU directory that came with the app and replaces it with the DKU directory packaged inside the module. Next, if the specializer has any code-structure transforms (CST), a script invokes those. When transform completes, a script invokes the compilation process. Finally, a script packages the resulting executables and sends the package, along with information about the specializer, the application, and the target hardware, to the Server's distributor.

The Server's distributor accepts executable bundles from specializer modules, and also listens for requests from clients running on end-platform hardware. To run an application on a given machine, a client on that machine is used to get the application. It sends information about the hardware and the desired application. The Distributor sends back the appropriate executable. The client unpacks the executable and makes it available to the OS to be run. The client may simply be a human using FTP, a script or a test harness.

10.7.3 Client on end-hardware

We chose to include a client on end-hardware to improve support for the search for the boundary between application and hardware. The search needs a broad base of applications; to encourage application development we provide "turn-key" infrastructure that automates the

entire toolchain, all the way to invoking an application on target hardware via a BLISS Client. The right of the top of Figure 10.2 shows that a client sends detailed information about its platform to the Specialization Server, potentially including cache sizes, number of cores, operating system settings, and so forth. The server hands the client the executable best suited to the hardware. Our sample code implements the client as a human using FTP and a web-page interface to the data base.

A more interesting client would be one that runs on a mobile device, as envisioned for the OMP (Open Media Platform) project [3] which supported this work on the BLISS framework. Here, the client requests the media-components required to play a particular content stream, as needed, and may even request according to resource usage and quality of experience offered.

10.8 Specializers and the Schedulers Inserted by Them

The heart of the BLISS framework is the specializers that produce the executable images.

There is only one formal requirement for specializers and schedulers: Specialization must be performed, and the code output from Specialization must produce the same result as with the sequential scheduler. Beyond this, they are free. For instance, they don't even have to use the library interface call to the scheduler; the specializer could perform a Code Structure Transform that turns the call to the Scheduler into an OS call to start the Kernels, and so on.

For our automated infrastructure, specializer modules often have two parts: a BLISS directory that contains pre-written scheduling + communication code, and a script that first replaces the serial BLISS directory, that came with the application source, with the hardware-specific one, and then compiles the result. The pre-written scheduler in the new BLISS directory makes calls to the D, K, U, etc implemented in the application, while the application calls the

scheduler. Such a specializer module does not modify the calls, so there are no compilation errors. Changing the contents of the BLISS directory has only changed the implementations of both directions of library call.

In the following sub-sections, we describe some details of our sample specializer modules and the schedulers packaged inside them, for each of our test hardware platforms.

10.8.1 Java specializer and scheduler for Shared Memory Multicore

Our sample specializer module for Java on shared memory multi-core machines has no Code Structure Transforms. It only removes the serial DKU directory and replaces it with one containing our pre-written scheduler for multi-core machines.

We didn't concentrate on performance, as our intent is to demonstrate how to make a scheduler, not to try to make an interesting one. The scheduler creates one worker-thread for each core, and communicates with it via a pair of one-way queues.

A separate `DKUScheduler` object is created in the application's main thread. Then its `scheduleAndPerformWorkOn` method is called and handed an `Object`. The method uses the `DividablePieceMaker` to turn the `Object` into an array of `DKUPiece` objects, then loops through the objects invoking the divider method of each one. It tells the divider to make the same number of pieces as the number of worker threads. Then it loops through the sub-pieces, sending each to a worker thread, round-robin. The worker threads take DKUPieces out of the queue, call the Kernel method of the DKUPiece, which produces results by side-effect, then put the piece into the return queue back to the scheduler. The scheduler calls the undivider method on the parent piece, passing it each sub-piece until the parent says all sub-pieces are accounted for. The scheduler then loops to the next DKUPiece in the array, and returns when all are done. There is significant room for scheduler improvement.

10.8.2 Java specializer for Heterogeneous Networks of Machines

The scheduler in this specializer module has two levels that run in separate JVMs: one scheduler is part of the application executable, and a second scheduler is in a stand-alone "worker" that runs on each machine in the heterogeneous collection. A worker accepts pieces from all applications running on the collection.

Two levels of division are performed, adaptively. The first level of division is performed by the application-scheduler. It reads a config file of the machines available, and makes enough pieces that it can hand each machine a number proportional to its processing power. The second division-level is performed inside each worker.

The worker running on a given machine is implemented specifically for that machine, so it potentially performs division and scheduling in its own way. Our implementation is for a collection of shared memory multicore machines, so the workers are all implemented the same, but they differ in the number of threads they schedule onto. The workers divide the pieces they receive, to end up with close to the same number of sub-pieces as there are hardware threads in the machine.

This demonstrates a useful property of the DKU interface, that division is re-entrant, so it can be performed repeatedly on sub-pieces and sub-sub-pieces, etc. The choice of further sub-division is left to the receiver of a piece.

10.8.3 C specializer for Cell BE Processor

The Cell BE hardware makes this specializer more interesting. The scheduler code is split into two parts, one part that runs on the two PPUs, and a second part that runs on the SPEs. As with the other schedulers, this is sample code to show how to make a scheduler for the Cell that uses the DKU interface, without special effort to gain performance.

The Cell has two different instruction sets inside it, one for the PPU, a second for the SPE, so the specializer must create two separate code bases and call two separate compilers (with help from the SDK). The functions from the application that execute on the SPEs are `unbundleInputs()`, `Kernel`, and `bundleResults()`, so these must be copied out of the application code-base and into the DKU directory where they are inserted into SPE code templates. We performed the copy and insertion by hand rather than writing scripts. It takes about 10 mins to specialize the generic source to run on the Cell.

In the C version of DKU, before calling the schedule function the first time, `schedulerInit()` is called. For the Cell, this init uploads the SPE code templates, with their inserted application functions, and starts them running. The SPEs then loop looking for work.

The PPU scheduler has the same architecture as the Java and C shared memory versions, with the difference lying in the worker threads. Rather than performing the work itself, the worker thread instead finds a free SPE and transfers the work to it.

This requires the worker thread to first call `bundleInputs`, then tell the SPE the address of the bundle. The SPE uploads the bundle and runs `unbundleInputs` in its local memory, hands the created DKUPiece to the Kernel, then calls `bundleResults` on the completed DKUPiece. The SPE then copies the result bundle to the PPU memory and notifies the worker threads. The worker that gets the notification pairs the returned result bundle with the original DKUPiece in PPU memory and runs `unbundleResults`, which moves data out of the bundle. After the data-moves, PPU memory looks just like the work had been performed locally.

The specializer has to handle machine details, such as the difference in pointer sizes between the PPUs, at 64bits, and the SPEs at 32 bits. It gets help from the application and the DKU standard.

Recall that the Kernel is defined to only touch data that is reachable from a DKUPiece, while `bundleInputs` is defined to gather all data a Kernel will touch from a given DKUPiece.

This hides global variables and shared variables from the scheduler code.

Native data size differences are hidden by `#define`s. The DKU.h file `#define`s standard size data types, for example `int32` and `uint8`. The application uses these for all data touched by the Kernel, instead of `int` and `char`. This allows the specializer to include a hardware-specific `DKU.h` file in its DKU directory. The `#defines` are written as part of the specializer, possibly using macros, and make the compilation performed inside the specializer use the right assembly instructions to be consistent with the defined size.

Pointer size differences and endian differences are handled by `#define`s as well, two for pointer sizes in local and remote memories, and two for the endianness in each. The bundling quad is written, by the application programmer, to use these `#define`s in the `DKU.h` file to construct a bundle with the right sizes and byte orderings for remote memory. Pointer arithmetic and normal arithmetic then function correctly inside a remote Kernel binary, which was compiled to a different ISA, without the application ever knowing what that ISA might be. Further, `if` statements in the bundling quad, that check the `#define` values to decide which rearrangements of data to perform, are optimized away, leaving only the operations required for the actual combination of ISAs to be compiled and run.

Lastly, the application should not use align statements in data structures that are copied by `bundleInputs`. If such statements are required, then `bundleInputs` is written to perform a primitive-by-primitive copy into the bundle. Meanwhile `unbundleInputs` is written to create a new data structure and copy the individual primitives into it. With this approach, the two compilers should correctly handle differences in alignment on the local and remote machines.

10.9 Experimental Results

We stress that the schedulers tested here are sample code, written to show how it is done, with no interesting features for performance. The results serve to show that the bidirectional interface pattern works; performance is placed by the BLISS interface into the hands of the specializer implementors and application implementors.

When measuring overhead, it can be subtle determining which overhead should be counted against the interface, and which is intrinsic to parallelism. We adopt this question as the defining test to determine what is overhead of BLISS vs overhead intrinsic to parallelism:

> "would this overhead occur in the application, if it were written by hand using the scheduling primitives available on the hardware?"

In most cases, the answer to the question is yes, the overhead would also occur in a hand-coded version, which makes the overhead intrinsic to parallelism. The one set of overheads that are clearly unique to using BLISS are the `call` instructions of the interface calls.

However, some overheads may be made worse by using BLISS, but by how much is implementation dependent. These are: loss of parallelism opportunities due to BLISS's choice of interface pattern; additional think time in a scheduler due to lack of needed information or lack of a needed "knob" to manipulate application-specific quantities; time lost in the barrier implied by the undivider when no independent work is available to overlap; and added overhead caused by the form of sending signals between the scheduler and a running Kernel that has been implemented to communicate.

We have no general solution to measure the worsening of these overheads resulting from the BLISS form. However, for some actions we measure times in the schedulers, which admits intuition about how efficient the BLISS code is.

In the experiments, we used a 2 core laptop, denoted "1x2", a 2 socket by 4 core each server, denoted "2x4", a 4 socket by 4 core each server, denoted "4x4", a heterogeneous

network of them connected by 100Mbit LAN, denoted "Het", and the Cell BE in a PS3. Of note, the heterogeneous network demonstrates the use of the re-entrant feature of the Divider, as mentioned in Sec 10.8.2.

Matrix Multiply in Java: Figure 10.5 shows efficiency on Matrix Multiply in Java, when run on three different multi-core machines, plus the heterogeneous collection of them. The break even size is around 100x100 double precision on the multicore machines. However, on the heterogeneous collection, the amount of data sent over the 100 Mbit LAN limits performance. Break even is around 200x200, where serial Kernel time is that on the machine the app is launched from. On matrix multiply, the comp-to-comm ratio grows linearly with matrix size, causing percent ideal speedup to increase linearly with matrix size.

The Slow-down for small matrix sizes can be avoided by the addition of an interface for execution-time prediction. When added, the schedulers will be able to avoid the parallelism overhead by instead calling the serial Kernel when a slow-down is predicted.

Hamiltonian Path in Java: Hamiltonian Path is used to demonstrate several things: the interface does evolve as needed; the interface admits high-efficiency; and complex problems can be expressed within the DKU interface. Hamiltonian Path is a complex problem, its running time is inherently not predictable, as far as is currently known, so the running-time of an input-graph in a serial thread is the base quantity, rather than size of the input-graph.

Table 10.1 shows the evolution of the interface: two input-graphs are run on a 4x4 core machine four times. Each time is with a different implementation of the scheduler: serial, one-time division, redivision, and redivision plus early termination. The dramatic changes in running time show the value of being able to incrementally add features to the interface. The fact that the same application source is run on all four versions of the scheduler demonstrates the point that "old" schedulers can be successfully used with "new" applications. This backwards-and-forwards compatible feature of the framework has practical value for researchers.

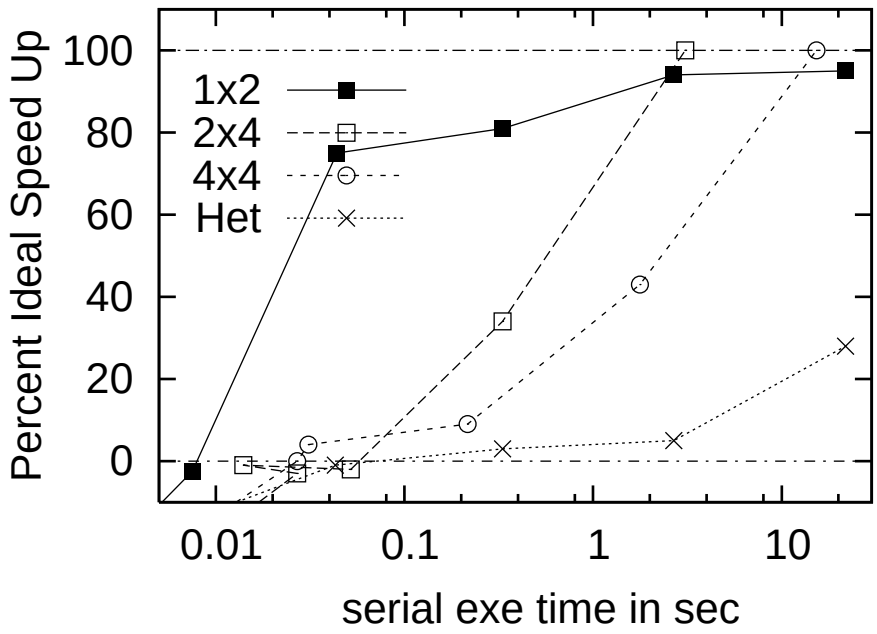

Figure 10.5: Matrix Multiply in Java: The percentage of the ideal speedup achieved on each of four machines vs the serialKernel execution time. Break-even exe time on a machine is found at the 0% point (serial exe time = parallel exe time). The size of matrix at successive points along one curve: 9x9, 81x81, 162x162, 324x324, 648x648, 1296x1296. ($PercentISU = \frac{\frac{Tser}{Tpar}-1}{p-1} \cdot 100$)

The odd timings are due to the nature of the search Kernel: it explores the search tree depth-first until it finds a solution, then stops. So the existence of a solution allows early termination, while on a graph with no solution, the full search tree must be explored. Hence, dividing the search tree of a graph can separate a solution into one piece, while the other pieces have no solution. The solution-containing piece finishes early while the others fully explore their sub-trees. This is why exe time increases from sequential to one-time divide: a previously hidden sub-tree with no solution is placed into its own piece, and all the other pieces wait for that one to complete.

This behavior drove the addition of the redivide and early-terminate interface extensions. In the "redivide" interface, the Kernel stops when signaled by the scheduler, and the work

Ham. Path	Sched:	Serial	One Div.	Rediv.	Rediv. + Early Term.
	Solution	140s	272s	43s	0.012s
	No Soln.	52s	21s	3.4s	3.3s

Table 10.1: Four versions of the scheduler, each on two input-graphs run on a 4x4 core machine. The top row shows running times for the graph that has a solution, while the bottom row is for a graph that has no Hamiltonian Path in it.

remaining in the piece is divided then handed out to idle workers. The table shows that, for this input-graph, redivision gives near linear speedup over one-time divide. The deviation from fully linear is due to working on the original pieces, that finished early. Notice that it would finish even faster if the long-running pieces could be stopped as soon as a solution is found. This is what the Early-terminate interface provides. Its use gives the surprising speedup seen in the top of the last column. However, when no solution exists in the input graph, the speedup remains linear as seen just below, in the last column of the second row.

Figure 10.6 shows efficiency: the same input-graphs are run on all the test machines, in a single thread first then on multiple threads. The input-graphs all have no solution, so the amount of work stays constant. The plot shows that the speedup approaches perfect speedup on all machines once the serial time is large enough. This shows that reasonable performance can be achieved even with very simple schedulers.

Table 10.2 shows Cell BE results for deblocking of a cell-phone size screen. All of the parallel configurations using the SPEs caused slowdown due to communication delays. As seen by the Kernel time vs Comm time, deblocking performs very little work on each byte transferred, so communication latency dominated. However, in a system with other sources of parallelism, the latency might be overlapped.

This illustrates the importance of the serial kernel. As seen here, not all sources of parallelism exposed in an application are exploitable on all hardware. The serial kernel's speedup of 2.3x over the 6 SPE configuration shows that the flexibility to decide at run-time whether to

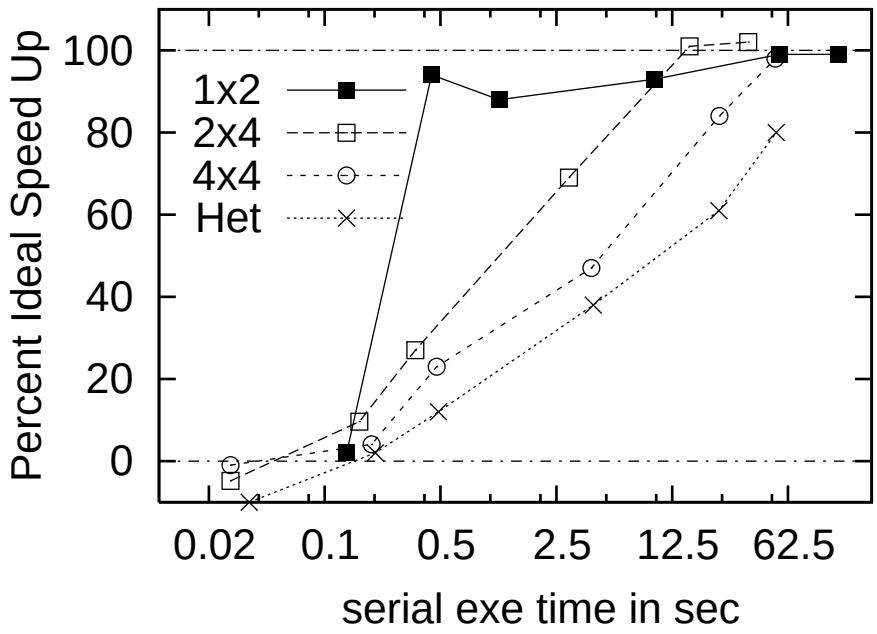

Figure 10.6: Hamiltonian Path in Java: The percentage of the ideal speedup achieved on each of the multicore machines vs the serialKernel execution time. All input graphs are no solution types. The Redivide + Early Terminate scheduler was used. Exe time is the same for serial and parallel execution at the 0% SU point.

exploit the parallelism can significantly affect performance.

These Cell results show how the framework can be useful. The hand-coded versions of H264 achieve parallel speedup, by performing multiple steps on the same data once it is in an SPE. To close the gap between hand-coding vs machine-independent source, greater flexibility is needed in automatically arranging the sources of parallelism exposed in the portable code. This framework helps in exploring how to express the needed flexibility, and in exploring how to write a specializer and a scheduler for the Cell that take advantage of that expressed flexibility. Productivity gains in doing that kind of research are the goal of the proposed framework.

H264 D.F. on Cell BE	Serial PPE	1 SPE	3 SPE	6 SPE
Total time	2.0ms	16ms	8.9ms	4.7ms
Kernel time	n/a	2.2ms	0.7ms	0.3ms
Comm time	n/a	13ms	7.6ms	4.2ms

Table 10.2: Cell results on H264 deblocking. Total time is for one frame of 320 x 200 pixels. Kernel time is the portion of that due the Kernel. Comm time is the portion due to the bundling quad plus queues plus DMA. The "missing" time was spent in the scheduling code on the PPU. "Serial" means the serialKernel was run on the PPU, while "X SPE" means the parallel version was run using that many SPEs.

10.10 Chapter Summary

We have shown a framework to support research on portable high performance parallelism. It uses the pattern of bi-directional library calls to successfully separate application code from hardware specific code, while enabling mix-and-match of applications to hardware. The hardware scheduler is called as a library function, and interacts with "knobs" that the application makes available as reverse-library functions.

We showed a bidirectional library interface for data parallelism, three applications instrumented with it, and results of automated specialization of the applications to shared memory multicore machines. The applications were written and debugged in a sequential IDE in Java and C; then the sources were specialized, automatically or by hand, to the Cell BE, to various multi-core, and to a heterogeneous collection of multi-core. The performance results show that this approach works, and anecdotal evidence of the time we spent on implementation of the specializers indicates that the proposed framework may be efficient and helpful for research on portable parallelism.

Chapter 11

VMS: A Second Approach to Productivity and Portability

Why need a second approach? BLISS has many positive properties to support the collective search, however it also has a few potential drawbacks. Among them: the source is what gets placed on the server, which is sensitive for companies; the logistics of gathering the specialization modules and redistributing them requires a trusted entity to manage the process, which may be problematic establishing and maintaining; a centralized server for distribution may be awkward in some segments like web and enterprise; the ability to reuse toolchain effort among specialization modules is not clear; the technical feasibility is not clear for creating languages that will pass through such a large variety of specialization modules without machine-specific build-errors.

Mitigation of potential negatives to BLISS: A non-profit could be established, which manages the collection and redistribution of specialization modules. Once going, its value will motivate benefitting companies to contribute to maintain it. With this in place, each developer

can maintain their own specialization server, and get updates from that entity. That mitigates concerns over intellectual property. For distribution, the specialization server could also be split into an install-image generator, which holds the source and lives with the developer, and a distribution back-end, which is centralized and collects from many developers. This scheme also allows web and enterprise to maintain their own local specialization server that only has the specialization modules needed for their hardware. However, reuse and build issues remain open.

What VMS provides that BLISS doesn't: VMS was created to ease implementation of the runtime for new parallel language constructs, and happens to also enhance reusability of toolchains and simplification of specialization and distribution. In contrast to BLISS, it is compatible with a decentralized distribution system, because specialization is divided into three distinct phases with VMS.

The dominant hardware abstraction is currently Threads. So some background on Threads is cogent, to analyze what about them should be avoided and what about them should be kept. This background motivates the original drive to create VMS. It's application as a framework to support the collective effort towards the triple challenge comes in the next chapter.

11.1 Threads: The Good, the Bad and the Ugly

Threads were invented to be virtual sequential processors back when there was only a single physical core and only a single physical memory array per machine. There were no issues with multiple copies in different memory arrays of the same logical location, nor were there performance issues of trying to minimize the communication between those memory arrays. As a result, the Thread model was not designed with these issues in mind and ends up being ill suited to multi-core processors with multiple caches.

What's Good about Threads: They provide a virtualization of the computation resources. As many virtual processors can be created as the programmer desires.

What's Bad about Threads: Their parallel performance is poor, because they block the language from controlling task placement. This prevents the specialization that minimizes communication. Instead, a Thread scheduler is blind to parallel tasks in the application and so assigns those tasks to cores blindly.

This is bad because the scheduler in the language's runtime is capable of keeping track of where it has placed tasks and what data each consumed and produced. So it can estimate the locations of data in the memory hierarchy. When it places a new task, it can search for a location close to the data that task will consume, and so choose one that reduces communication of the data. This improves performance and also reduces energy per task.

So, it's Bad that Threads prevent the language's runtime from implementing such a scheduler. A replacement has to support the language in performing such minimization of communication.

What's Ugly about Threads: They are notoriously difficult to program with. The Thread model includes synchronization constructs as part of the model. The exact constructs vary between Thread implementations: semaphores, locks, monitors, mutexes, condition variables, and so on. However they all are similarly difficult to write correct programs with, and don't provide what the specialization process needs to identify parallel tasks, the constraints on them, how to resize them, et cetera.

In more detail, Threads are Ugly because the mental model of a thread often changes when the programmer thinks about different aspects of a program. A thread is sometimes thought of as a virtual processor that communicates with other processors, such as when queues

are used to communicate between threads. Secondly, other times a thread is thought of as an animator, that during some portions of physical time actively animates code (or a virtual processor), and during other physical time is dormant. Such an animator has corresponding animator state such as the pthread data-structure or the Java Thread class, with which the application can control the animation (note that this control over animation is intrinsically part of the scheduling process, so exposing it in the application interferes with specialization). Thirdly, at other times a thread is thought of as a stateless time-line, especially when looking at a piece of code, and imagining the interleavings of the time-lines through that same code, in order to figure out where locks are needed to control the interleavings.

As a result, the programmer's mind has to constantly switch gears, jumping between mental models. Worse, the interleavings model requires the programmer to think of *all possible interleavings* in order to write correct code. This is an exceptionally difficult task.

So, it's Ugly that Threads have a fixed set of parallelism constructs, which are difficult to use and don't support specialization.

What's Better? The ideal hardware abstraction would have morphable parallelism construct behavior, rather than having fixed behavior. Dozens to hundreds of families of alternative parallel constructs have been researched. Examples of families include Threads (for legacy code); Actors [48, 8]; Components [56]; process calculi [49, 61]; coordination languages [33]; and new ones continually invented. So a replacement for Threads should support all of them.

What a replacement to Threads should look like: Putting this all together, a replacement hardware abstraction should:

- morph parallelism construct behavior, to any desired constructs
- morph scheduling behavior, to be custom to an application or language, allowing them to

control placement of tasks onto computation resources

- allow equal access to hardware to all parallelism construct implementations (interrupts, scheduling hardware-accelerators and so on)
- allow the parallelism constructs and the scheduler to work together

11.2 VMS

VMS embodies all the points of the ideal hardware abstraction. It allows plugins to morph the behavior of the base hardware abstraction, letting any desired parallelism construct be plugged in, and any scheduler implementation. Depending on the VMS implementation, the constructs and scheduler have full control over the lowest level hardware mechanisms.

VMS will be explained in four levels, from abstract definition down to implementation details.

First, the theoretical definition is given, in Section 11.3, to lay the mental framework. Then it is related the elements of the multi-core implementation, in Section 11.4. How to use a VMS-based parallelism library comes next, showing the application code point of view, in Section 11.5. It also gets related back to both the abstract model and the internal elements. Lastly, implementation details appear at the end of Section 11.5, with the implementation of the plugin for the send-receive parallel construct. To put show that VMS provides good plugin-creation productivity and competes for low overhead, measurements appear in Section 11.6

Virtual Processor (VP) Definition: To avoid the confusion associated with the terms "thread" and "task" this chapter will use the term *virtual processor* (VP), which is defined as state in combination with the ability to animate code or *an additional level of virtual processors*. The state consists of a stack with its contents, a program counter, a pointer to top of stack, and a pointer to the current stack frame.

11.3 Definition of VMS

This section begins with an intuitive overview, and adds details in the following sub-sections.

Intuitive Overview: VMS is concerned primarily with time and guarantees about it. This is because what parallelism constructs do is control how the time-lines of different virtual processors intersect. They also guarantee relations of time lines to hardware events.

As an example, consider a program that writes into a data structure in one time-line, then calls a `send` construct, meanwhile in a different time-line it calls the `receive` construct then reads the data structure. The constructs should guarantee that all data written before the `send` is readable in the other time-line after the `receive`. VMS provides primitive guarantees, which plugged-in code builds upon to provide such higher-level guarantees.

To support parallelism constructs, VMS provides: primitive operations to create and suspend VPs; a way for plugged-in code to control when each VP is (re)started; and time-related guarantees. These are enforced on all hardware, be it shared memory or distributed, with strong memory consistency or weak.

Definition in Three Parts: We give the abstract definition in three parts: a definition of the elements of a VMS computation system; a definition of time and the key VMS guarantee; and a definition of virtual processor scheduling states and transitions between them.

The definition we give is for VMS *with plugins present.* Hence, it covers the behavior of all possible parallelism constructs implementable with VMS. The Master mentioned in the definition is an abstract entity, with a plugin present. In practice, this Master entity is implemented as part of a core VMS, and plugins later added. This VMS-core is the hardware abstraction. It hides the physical hardware behind an interface that creates virtual processors and enforces

well-defined time-behavior.

11.3.1 The Elements of a VMS Computation System

- A VMS program has multiple VPs, which are Slaves, each with an independent time-line.
- A schedule of Slaves is generated by a Master entity, from within separate time-line(s).
- A schedule is defined as the set of points at which VPs are (re)animated.
- All semantic parallelism behavior is invoked via communication with the Master.
- Communication with the Master happens by using a model-provided primitive, which causes *voluntary* suspension of the program's VP.

What is important here is: that the choice of which VP is animated, at which point, happens in a separate time-line; and that the VPs suspend voluntarily for each parallelism construct. This means that *scheduling is separated from the application code*, the key point.

The Master entity appears to be a single entity to the slaves, but may be implemented by multiple Master VPs hidden inside the VMS implementation.

VPs use the Master as an intermediary to: semantically communicate with each other; cause creation of new program VPs; and to influence re-animation of VPs. As a subtlety, notice that hardware mechanisms, such as coherent shared memory, allow communication to take place that is not visible to the parallelism constructs. Parallelism constructs must be separately called in order to make use of shared variable communication safe.

Definitions: VMS is intended only for hardware systems that consist of processing elements connected by communication. We define a memory-space to be a processing element, albeit without the ability to transform data. We define a *physical core* to be a processing element that *does* transform data, and require that it execute a sequential stream of instructions. We define a program-time as the sequence of instructions animated by a Slave VP (which is eventually

animated by a physical core). A Slave VP has associated *scheduling state* that, among other things, relates to how its program-time progresses relative to physical time on the cores.

11.3.2 Time in VMS

- VMS has three levels of time: *Program time*, *Master time*, and *Virtual time*.
- Program time is local to a Slave VP, measured in instruction executions.
- Master time is hidden from the program and is independent from all Program times.
- Virtual time is the ordered set of changes in scheduling state of Slave VPs.

What is most important here is that Virtual time defines a global sequential ordering. This ordering is consistent with the key VMS guarantee (given below), and each point in it is computed within Master time.

Also, the independence between program times and master time has subtle advantages. It enables elegant enforcement of the VMS guarantee, and implementation simplifications that become clear after gaining deep implementation knowledge.

In VMS, each event relevant to parallel semantics is tied to a transition of the state of a Slave VP. This means that implementing the behavior of parallel semantics is equivalent to controlling the order of transitions of state of virtual processors.

Definitions: The stream of instructions in a given program-time is broken into a number of *trace-segments*, separated by suspension points. Each trace-segment is animated by a single physical core, but not necessarily the same core as animated the other trace segments. A suspend point is created by a Slave VP executing the "suspend" primitive provided by VMS. A suspend point has no duration in program time, but has distinct start and end points in virtual time. The end-suspension points of two different program times can be tied together within virtual time, which is called a *tie point* and has special significance to parallel constructs. The physical-time

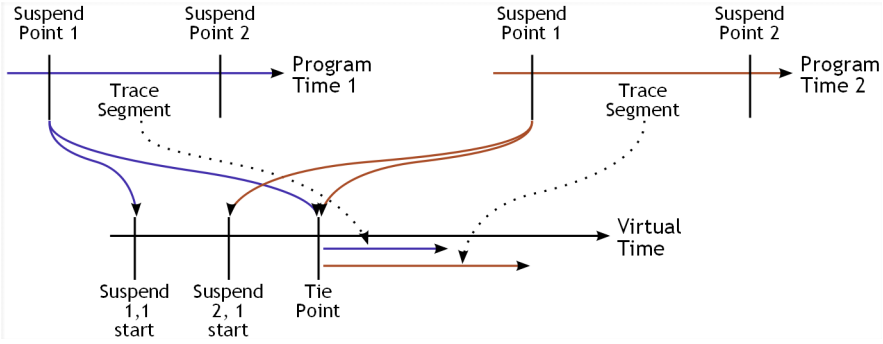

Figure 11.1: Mapping program time onto Virtual time. The Master controls creation of new program time lines, and ending suspend points. Here, it has ended two suspend points at a common tie-point.

of a core has no relationship to any program time, except for the various time-guarantees in this definition of VMS.

Relating time-lines to each other: Figure 11.1 illustrates how trace-segments relate to suspend points, and map onto virtual time. A trace segment starts in virtual time where suspend is ended, as seen. In fact, the two trace segments shown have a common start-point within virtual time. This is because the parallelism semantics chose to start them at the same point – this is what a tie point is. A key note is that the lengths in virtual time have no relation to the lengths in program-time. The only defined feature is that those two trace-segments have a common start-point in virtual time. This means that the two suspend points are considered to be tied together.

The Key VMS guarantee: Being tied together means that all physical events that can be observed by both program-times are covered by the key VMS guarantee: any events triggered before the common suspend point in one program time are guaranteed visible in the other program time after the common suspend point. They *might* be visible before, but it's not guaranteed. In addition, events triggered after the common suspend point in one are guaranteed

not visible before the common suspend point in the other. *This two-part guarantee can be considered the heart of VMS.*

Intuitively, a tie-point separates before it from after such that tied program times agree (illustrated with code in Section 11.5 Figure 11.5). But the subtlety is that events triggered before the tie-point, *might* be visible to the other before, and ones triggered after *might not* be visible to the other after – physical events triggered before are only guaranteed visible *after* the tie point, and events after are only guaranteed *not* visible *before* the tie point.

This is a form of bounded non-determinism. The pattern of suspension end-points determines which trace-segments overlap in Virtual time, and events triggered in one might be visible in overlapped ones. But no guarantees cover these. If one segment tries to observe, it will see events triggered by overlapped segments in non-deterministic order.

The VMS implementation defines which physical events are covered by the key VMS guarantee (reads/writes, network communication, DMA, I/O).

Globally consistent sequential order: VMS maps suspend-start, suspend-end, and hence tie-points, to a globally-consistent sequential order in Virtual time. This enables one of VMS's key benefits: sequential algorithms for parallel constructs.

Tie points define parallel behavior, so an implementation of how to choose tie points equals an implementation of parallel constructs. The Master chooses tie-points, so plugging code to choose tie-points into the Master equals plugging in parallel constructs.

11.3.3 Scheduling State

Scheduling state is used in VMS to organize internal activity, for enforcing the guarantees.

- VPs have three scheduling states: *Animated*, *Blocked*, *Ready* (Figure 11.2).

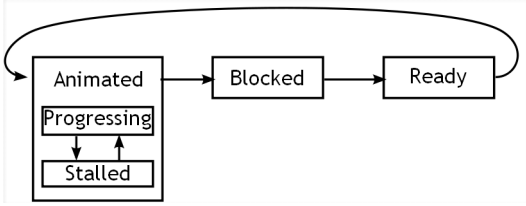

Figure 11.2: Scheduling states of a slave VP in the VMS model.

- VPs in Animated are *allowed* to advance program-time with *local* physical time.
- VPs in Blocked and Ready do not advance their program-time.
- Animated has two physical states: *Progressing* and *Stalled*.
- VPs in Progressing advance program-time with local physical time, those in Stalled do not (allowing non-semantic suspend).
- Scheduling states are defined in Virtual time only.
- Progressing and Stalled are defined in local physical time only (invisible in Virtual).

Some important points: 1) only VPs Animated can trigger physical events that are seen in other program time-lines; 2) the distinction between Blocked vs Stalled is that a VP has to explicitly execute a VMS primitive operation to enter Blocked, making it part of the semantics of parallelism constructs. In contrast, Stalled happens invisibly, with no effect on semantic behavior. It is due to hardware events hidden inside VMS, such as interrupts.

The Ready state is used to separate the parallelism-construct behavior from the scheduling behavior. It acts as a "staging area" for scheduling. VPs placed into this state are *allowed* to be animated, then the scheduler decides when and where.

A subtle but illustrative point is that actions *outside* a given program time cause the VP to transition Blocked→Ready, which contrasts to lock algorithms like spin-locks or Dijkstra's, where the concurrency-related behavior takes place *inside* program time.

Transition Between Slave Scheduling States: - VPs transition states as shown in Figure 11.2.

- Animated→Blocked is caused by a Slave VP executing the Suspend VMS primitive.
- Blocked→Ready is determined by the semantics implemented in the plugin.
- Ready→Animated is determined by the scheduler in the plugin.
- Transitions in scheduling state have a globally consistent order in Virtual time.

The parallelism primitives executed by a program do not control change in scheduling states. They merely communicate messages to the Master, via a VMS supplied primitive. Inside the Master, the plugin's parallelism construct implementation processes the message. Based on that, it performs changes in state from Blocked→Ready, creates new VPs, and dissipates existing VPs. Most communication from Slave to Master requires the VP to suspend when it sends the message. A few messages, like creating new Slave may be sent without suspending.

The suspend primitive decouples local physical time from Virtual time. Execution causes immediate transition to Stalled in physical time, then the Master performs Animated→Blocked, fixing that transition in Virtual time. The only relationship is causality. This weak relation is what allows suspension-points to be serialized in Virtual time, which in turn is what allows using sequential algorithms to implement parallelism constructs.

11.3.4 Plugins

The Master entity has two parts, a generic core part and a plugin (Figure 11.3). The core part of the Master is implemented as part of VMS-core. The plug-in supplies two functions: the communication-handler and the scheduler, both having a standard prototype. The communication-handler implements the parallelism constructs, while scheduler assigns VPs to cores.

An *instance* of a plugin is created as part of initializing an application, and the instance

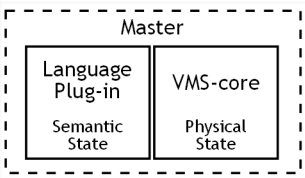

Figure 11.3: The Master has been split into a generic core and a language-specific plug-in. The core encapsulates the hardware and remains the same across applications. The plug-in is part of the parallelism-construct implementation. It is loaded separately onto the hardware and linked to the application when run.

holds the semantic and scheduling state for that run of the application. This state, combined with the virtual processor states of the slaves created during that application run, represents progress of the work of the application. For example, multi-tasking is performed simply by the Master switching among plug-in instances when it has a resource to offer to a scheduler. The parallelism-semantic state holds all information needed to resume (hardware state, such as TLB and cache-tags is inside VMS-core).

11.4 Internal Workings of Our Implementation

We name the elements of our example implementation and describe their logical function, then relate them to the abstract model. We then step through the operation of the elements.

Elements and Their Logical Function: As illustrated in Figure 11.4, our VMS implementation is organized around physical cores. Each core has its own *master virtual-processor*, `masterVP`, and a *physical-core controller*, which communicate via a set of scheduling slots, `schedSlot`. The Master in the abstract definition is implemented by the multiple `masterVP`s plus a particular plugin instance with its shared parallelism-semantic state (seen at the top).

On a given core, only one of: the core-controller, `masterVP`, or a slave VP, is ani-

mated at any point in local physical time. Each `masterVP` animates the same function, called `master_loop`, and each slave VP animates a function from the application, starting with the top-level function the slave is created with, and following its call sequence. The core controller is implemented here as a Linux pthread that runs the `core_loop` function.

Switching between VPs is done by executing a VMS primitive that suspends the VP. This switches the physical core over to the controller, by jumping to the start of the `core_loop` function, which chooses the next VP and switches to that (switching is detailed in Section 11.5 Figure 11.7).

Relation to Abstract Model: We chose to implement the Master entity of the model by a set of `masterVP`s, plus plug-in functions and shared parallelism-semantic state. What we call VMS-core consists of this implementation of the Master, plus the core-controllers, plus the VMS primitive libraries, for creating new VPs and dissipating existing VPs, suspending VPs, and communicating from slave VP to Master. In Figure 11.4, everything in green is part of VMS-core, while the plugin is in red, and application code appears as blue, inside the slave VP.

Virtual time in the model is implemented via a combination of four things: a `masterLock` (not shown) that guarantees non-overlap of `masterVP` trace-segments; the `master_loop` which performs transition Animated→Blocked; the `comm_handler_fn` which performs Blocked→Ready and the `scheduler_fn` which performs Ready→Animated. Each state transition is one step of Virtual time; is guaranteed sequential by the non-overlap of `masterVP` trace segments; and is global due to being in parallelism-semantic state that is shared (top of Figure 11.4).

Transitions Progressing⇄Stalled within the Animated state are invisible to the parallelism semantics, the Master, and Virtual time, and so have no effect on the elements seen.

Steps of Operation: The steps of operation are numbered, in Figure 11.4. Taking them in order, 1) `master_loop` scans the scheduling slots to see which ones' slaves have sus-

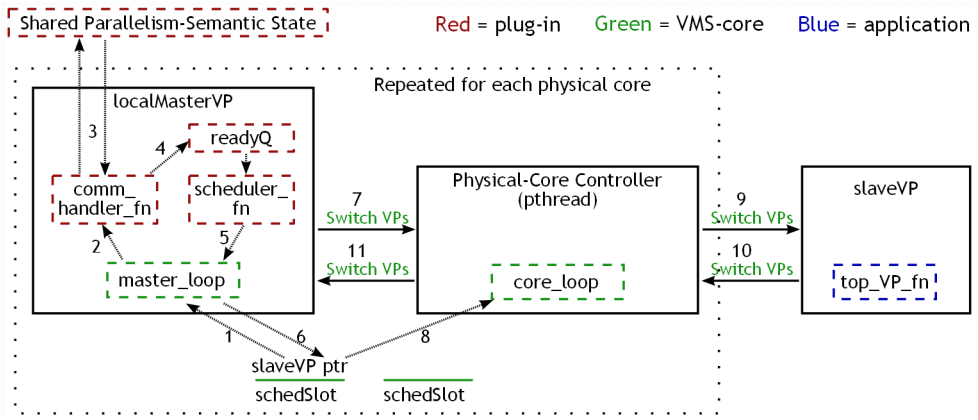

Figure 11.4: Internal elements of our example VMS implementation

pended since the previous scan. 2) It hands these to the `comm_handler_fn` plugged in (which equals transition Animated→Blocked). 3) The VP has a request attached, and data in it causes the `comm_handler_fn` to manipulate data structures in the shared parallelism-semantic state. These structures hold all the slaves in the blocked state (code-level detail in Figure 11.8, Section 11.5). 4) Some requests cause slaves to be moved to a `readyQ` on one of the cores (Blocked→Ready). Which core's `readyQ` receives the slave is under plugin control, determined by a combination of request contents, semantic state and physical machine state. 5) During the scan, the `master_loop` also looks for empty slots, and for each calls the `scheduler_fn` plugged in. It chooses a slave from the `readyQ` on the core animating `master_loop`. 6) The `master_loop` then places the slave VP's pointer into the scheduling slot (Ready→Animated), making it available to the `core_loop`. 7) When done with the scan, `masterVP` suspends, switching animation back to the `core_loop`. 8) `core_loop` takes slave VPs out of the slots, then 9) switches animation to them. 10) When a slave self-suspends, animation returns to the `core_loop` (detail in code in Figure 9), which picks another, until 11) all slots are empty and the `core_loop` switches animation to the `masterVP`.

Enabling sequential implementation of parallelism semantics: All of that happens on each core separately, but in this particular implementation we use a central `masterLock` to ensure that only one core's `masterVP` can be active at any time. This guarantees non-overlap of trace-segments from different `masterVP`s, allowing the plugins to use sequential algorithms, without a performance penalty, as verified in Section 11.6.

Relating this to the abstract model: the parallelism-semantic behavior of the Master is implemented by the communication handler, in the plugin. It thus runs in the Master time referred to, in the model, in Section 11.3. Requests are sent to the Master by self-suspension of the slaves, but sit idle until the other slaves in the scheduling slots have also run. This is the passive behavior of requests that was noted in Section 11.3, which allows the `masterVP`s to remain suspended until needed. This in turn enables the `masterVP`s from different cores to be non-overlapped. It is the non-overlap that enables the algorithms for the parallelism semantics to be sequential.

11.5 Code Level View

To relate the abstract model and the internal elements to application code and parallelism-library code, we give code snippets that illustrate key features. We start with the application then work down through the sequence of calls, to the plugin, using our SSR [41] parallelism-library as an example.

In general, applications are either written in terms of a full custom language that has its own syntax, or a base language with a parallelism library, which is often called an *embedded language*. Our demonstrators, VCilk [41], Vthread, and SSR, are all parallelism libraries. A full custom language would follow the standard practice of performing source-to-source transform, from custom syntax into C plus parallelism-library calls.

Creating a new processor:

```
newProcessor = SSR__create_procr( &top_VP_fn, paramsPtr, animatingVP );
```

prototype for the top level function:

```
top_VP_fn( void *parameterStrucPtr, VirtProcr *animatingVP );
```

handing animating VP to parallelism constructs:

```
SSR__send_from_to( messagePtr, animatingVP, receivingVP );
messagePtr = SSR__receive_from_to( sendingVP, animatingVP );
```

Figure 11.5: Application code snippets showing that all calls to the parallelism library take the VP animating that call as a parameter.

SSR: SSR stands for Synchronous Send-Receive, and details of its calls and internal implementation will be given throughout this section. It has two types of construct. The first, called *from-to* has two calls: `SSR_send_from_to` and `SSR_receive_from_to`, both of which specify the sending VP as well as the receiving VP. The other, called *of-type* also has two calls: `SSR__send_of_type_to` and `SSR__receive_of_type`, which allow a receiver to accept from anonymous senders, but select according to type of message.

Application View: Figure 11.5 shows snippets of application code, which use the SSR parallelism library. The most important feature is that all calls take a pointer to the VP that is animating the call. This is seen at the top of the figure where slave VP creation takes a pointer to the VP asking for creation. Below that is the standard prototype for top level functions, showing that the function receives a pointer to the VP it is the top level function for.

The pointer is placed on the stack by VMS when it creates the VP, and is the means by which the application comes into possession of the pointer. This animating VP is passed to all library calls made from there. For example, the bottom shows a pointer to the animating VP placed in the position of sender in the `send` construct call. Correspondingly, for the `receive` construct, the position of receiving VP is filled by the VP animating the call.

Relating these to the internal elements of our implementation, the `animatingVP` sus-

```
void * SSR__receive_from_to( VirtProcr *sendVP, VirtProcr *receiveVP )
 { SSRSemReq  reqData;
   reqData.receiveVP = receiveVP;
   reqData.sendVP    = sendVP;
   reqData.reqType   = receive_from_to;
   VMS__send_sem_request( &reqData, receiveVP );
   return receiveVP->dataReturnedFromRequest;
 }
```

Figure 11.6: Implementation of SSR's receive_from_to library function.

pends inside each of these calls, passing a request (generated in the library) to one of the `masterVP`s. The `masterVP` then calls the `comm-handler` plugin, and so on, as described in Section 11.4.

For the `SSR__create_processor` call, the comm-handler in turn calls a VMS primitive to perform the creation. The primitive places a pointer to the newly created VP onto its stack, so that when `top_VP_fn` is later animated, it sees the VP-pointer as a parameter passed to it. All application code is either such a top-level function, or has one at the root of the call-stack.

The send and receive calls both suspend their animating VP. When both have been called, the communication handler pairs them up and resumes both. This ties time-lines together, invoking the VMS guarantee. Both application-functions know, because of the VMS guarantee (Section 11.3), that writes to shared variables made before the send call by the sender are visible to the receiver after the receive call. This is the programmer's view of tying together the local time-lines of two different VPs, as defined in Section 11.3.

Concurrency-Library View: A parallelism library function, in general, only creates a request, sends it, and returns, as seen below. To send a request, it uses the combined request-and-suspend VMS primitive that attaches the request then suspends the VP. The primitive requires the pointer to the VP, to attach the request and to suspend it.

In Figure 11.6, notice that the request's data is on the stack of the virtual processor

that's animating the call, which is the `receiveVP`. The `VMS__send_sem_request` suspends this VP, which changes the physical core's stack pointer to a different stack. So the request data is guaranteed to remain undisturbed while the VP is suspended.

Figure 11.7 shows the implementation of the VMS suspend primitive. As seen in Figure 11.4, suspending the `receiveVP` involves switching to the `core_loop`. In our implementation, this is done by switching to the stack of the pthread pinned to the physical core and then jumping to the start-point of `core_loop`.

This code uses standard techniques commonly employed in co-routine implementations. Tuning effort spent in `core_loop` is inherited by all applications.

```
VMS__suspend_procr( VirtProcr *animatingVP )
 { animatingVP->resumeInstrAddr = &&ResumePt;
    //GCC takes addr of label
   animatingVP->schedSlotAssignedTo->isNewlySuspended = TRUE;
    //for master_loop to see
   <assembly code stores current physical core's stack reg into
    animatingVP struct>
   <assembly code loads stack reg with core_loop stackPtr, which was
    saved into animatingVP>
   <assembly code jmps to core_loop start instr addr, which was also
    saved into animatingVP>
 ResumePt:
   return;
 }
```

Figure 11.7: Implementation of VMS suspend processor. Re-animating the virtual processor reverses this sequence. It saves the `core_loop`'s resume instr-addr and stack ptr into the VP structure, then loads the VP's stack ptr and jmps to its `resumeInstrAddr`.

Plugin View: SSR's communication handler dispatches on the `reqType` field of the request data, as set by the `SSR__receive_from_to` code. It calls the handler code in Figure 11.8. This constructs a hash-key, by concatenating the from-VP's pointer with the to-VP's pointer. Then it looks-up that key in the hash-table that SSR uses to match sends with receives, which is in the shared semantic state seen at the top of Figure 11.4 in Section 11.4.

```
handle_receive_from_to( VirtProcr *requestingVP, SSRSemReq *reqData,
 SSRSemEnv *semEnv )
 { commHashTbl = semEnv->communicatingVPHashTable;
   key[0] = reqData->receiveVP;
   key[1] = reqData->sendVP; //send uses same key
   waitingReqData = lookup_and_remove( key, commHashTbl );//get waiting req
   if( waitingReqData != NULL )
    { resume_virt_procr( waitingReqData->sendVP );
      resume_virt_procr( waitingReqData->receiveVP );
    }
   else
      insert( key, reqData, commHashTbl );
       //receive is first to arrive, make it wait
 }
```

Figure 11.8: Pseudo-code of communication-handler for `receive_from_to` request type. The `semEnv` is a pointer to the shared parallelism-semantic state seen at the top of Figure 11.4.

The most important feature in Figure 11.8 is that both send and receive will construct the same key, so will find the same hash entry. Whichever request is handled first in Virtual time will see the hash entry empty, and save itself in that entry. The second to arrive sees the waiting request and then resumes both VPs, by putting them into their `readyQ`s.

Access to the shared hash table can be considered private, as in a sequential algorithm. This is because our VMS-core implementation ensures that only one handler on one core is executing at a time.

11.6 Results

Setup: We implemented blocked dense matrix multiply with right sub-matrices copied to transposed form. We ran on a 1 socket by 4 core Core2Quad 2.4Ghz chip.

Implementation-Time: As shown in Table 1, time to implement the three parallel libraries averages 2 days each. As an example of productivity, adding nested transactions, parallel singleton, and atomic function-execution to SSR required a single afternoon, totaling

Table 11.1: Person-days to design, code, and test each parallelism library. L.O.C. is lines of (original) C code, excluding libraries and comments.

	SSR	Vthread	VCilk
Design	4	1	0.5
Code	2	0.5	0.5
Test	1	0.5	0.5
L.O.C.	470	290	310

less than 100 lines of C code.

Execution Performance: Performance of VMS is seen in Table 11.2. The code is not optimized, but rather written to be easy to understand and modify. The majority of the plugin time is lost to cache misses because the shared parallelism-semantic state moves between cores on a majority of accesses. Acquisition of the master lock is slow due to the hardware implementing the CAS instruction.

Existing techniques will likely improve performance, such as localizing semantic data to cores, splitting malloc across the cores, pre-allocating slabs that are recycled, and pre-fetching. However, in many cases, several hundred nano-seconds per task is as optimal as the applications can benefit from.

Head to Head: The VMS implementation of the `spawn` and `sync` constructs is compared against Cilk 5.4, on the top in Table 11.3, which shows that the same application code has similar performance. For large matrices, Cilk 5.4's better use of the memory hierarchy achieves 23% better performance. However, for small matrices, VCilk is better, with a factor 2 lower overhead. Cilk 5.4 does not allow controlling the number of spawn events it actually executes, and chooses to run smaller matrices sequentially, limiting our comparison.

When comparing to pthreads, the VMS based implementation has more than an order of magnitude better overhead per invocation of mutex or condition variable functionality, as seen on the bottom of Table 11.3. Applications that inherently have short trace segments will

Table 11.2: Cycles of overhead, per scheduled slave. "comp only" is perfect memory, "comp + mem" is actual cycles. "Plugin-concur" only concurrency requests, "plugin-all" includes create and malloc requests. Two significant digits due to variability.

		comp only	comp +mem
VMS Only	`master_loop`	91	110
	switch VPs	77	130
	(malloc)	160	2300
	(create VP)	540	3800
Language:			
SSR	plugin – concur	190	540
	plugin – all	530	2200
	lock		250
Vthread	plugin – concur	66	710
	plugin – all	180	1500
	lock		250
VCilk	plugin – concur	65	260
	plugin – all	330	1800
	lock		250

synchronize often and benefit the most from Vthread.

11.7 Summary

VMS is an alternative to the Thread hardware abstraction that makes it quick to implement parallelism constructs, that have control over placement of tasks, and full access to the bare hardware at the lowest level. It can be thought of as splitting the scheduler open to accept new synchronization constructs and custom scheduler, supplied in the form of a plugin. The new parallelism constructs are implementable using sequential algorithms, within a matter of days, while maintaining hundred nano-second level overhead per concurrency operation.

Table 11.3: On top, exe time in seconds for MM. Below, overhead for pthread vs Vthread. First column is cycles for perfect memory and second is total measured cycles. pthread cycles are deduced from round-trip experiments.

Matrix size	Lang.	sec.
81x81	Cilk	0.017
	VCilk	0.008
324x324	Cilk	0.13
	VCilk	0.13
648x648	Cilk	0.71
	VCilk	0.85
1296x1296	Cilk	4.8
	VCilk	6.2

operation	Vthread		pthread	ratio
	comp only	total		
mutex_lock	85	1050	50,000	48:1
mutex_unlock	85	610	45,000	74:1
cond_wait	85	850	60,000	71:1
cond_signal	90	650	60,000	92:1

Chapter 12

The Use of VMS as a Framework to Collectively Meet the Triple Goal

Given the understanding of VMS from the previous chapter, this chapter goes on to show how it can be used to support a collective effort towards solving the triple goal of productivity, portability, and adoptability of parallel programming. VMS takes the role of integrating the various efforts without the need for coordination among research groups.

To solve the triple challenge, a highly productive language must be found that supplies specialization what it needs, while having a framework to perform the specialization to various target platforms, and do both in a way that fits the needs and constraints of at least one software segment, without very many barriers, nor any high barriers, to adoption.

VMS supports the search for such a language by simplifying the creation of new languages, as shown in the previous chapter. However, the question remains how does VMS support portability? And, once that is seen, what barriers does the proposed scheme present to the various software segments?

As discussed in part 1, specialization happens after an application completes develop-

ment. It needs information from the application to define tasks, control their size, predict their data footprint, and know their communication to other tasks. VMS breaks specialization into three distinct phases.

In review, a system for portability should have a means to identify language and application constructs and package this information into a standard format. Also, a hardware abstraction should be provided that accepts such information and uses it to make high-quality decisions: about task creation, sizing, and placement.

12.1 Three-step specialization

On nomenclature, *task* is defined as a 3-tuple: ¡animation event, collection of code animated, collection of information the code is animated upon¿. The phrase "create a task" means create a combination of code plus data *with the intent to animate.*

VMS makes specialization happen in three steps, as illustrated in Figure 12.1.

First, in the top box, the toolchain extracts task information useful to the scheduler and packages it into the binary. This specializes the source to the plugin's interface. Second, in the middle, the scheduler in the plugin retrieves the info and uses it to make scheduling decisions. This is specializing the binary to the hardware abstraction. Third, at the bottom, the VMS-core implementation hides hardware details behind the interface. This is specializing the hardware abstraction to the hardware.

VMS makes specialization happen in three steps, as illustrated in Figure 12.1.

The combination of plugin plus VMS has the same function that the instruction set had back in the sequential days – it provides a standard hardware abstraction. VMS has the advantage that the abstraction can be chosen separately for each binary, by choosing the plugin.

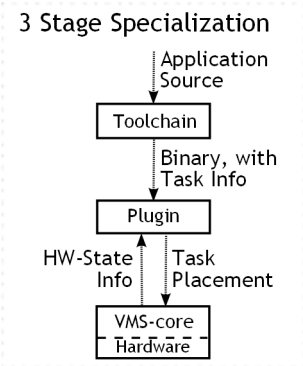

Figure 12.1: Specialization occurs in three places.

Isolating parallelism in the toolchain: The toolchain is broken into two sections: one for parallelism, the other for sequential. The parallelism portion extracts the task and language information needed to make high quality scheduling decisions. The sequential specializes individual functions to the sequential cores. The parallelism portion remains constant across hardware, only the sequential portion changes when the instruction-set of the target hardware changes (in effect, this moves a portion of the compiler into an install-time, or load-time, or run-time component that completes binary specialization).

One possible way to achieve this is to make the parallel portion transform the original source to C code, with embedded calls to the parallelism constructs. Also during this source-to-C-plus-lib-call transform, the task information is packaged into functions in some way. The resulting C-plus-lib-calls source is then compiled with a sequential C compiler to make a binary, as depicted in Figure 12.2.

Meanwhile, the plugins for that language know the names of the library functions the task information has been packaged into. Hence, when the binary is linked to a plugin at load time, the task-info functions within the binary become available to the plugin. They are called during the run to extract the information.

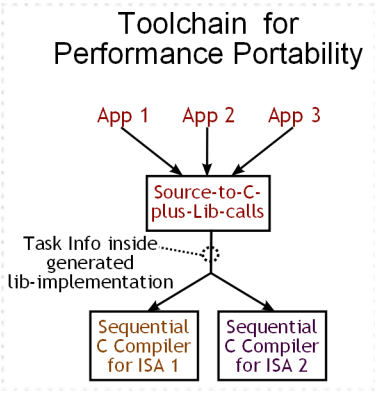

Figure 12.2: The toolchain is split into parallel and sequential portions. The parallel portion encodes task information as the bodies of special "information-carrier" functions, which the plugin later calls to retrieve the info.

This scheme allows existing, unmodified, sequential compilers to be used to pass the information along, inside of a standard binary format. It also separates the parallel and sequential portions of the tool chain cleanly, so only the relatively simple sequential C compiler changes with hardware. This scheme also leaves the language designers in charge of deciding the nature and definition of the information carrying functions, so it can evolve rapidly, while remaining compatible with the infrastructure. This accelerates the collective search.

These properties enable reuse of the same parallel portion of the toolchain across chips, which is especially valuable in the embedded market. With the inclusion of real-time aspects, such as latency bounds and deadlines, this could dramatically speed up time to market and reduce cost of introduction of new embedded chips.

Focus Area: This part of the proposal may run into difficulties in practice. The details of architectures like vector units and GPUs are too fine-grain for typical run-time scheduling, and may require multi-versioned binaries [31, 22] or binary optimizers [27, 73] or PetaBricks style adaptability. It will require extensive toolchain research by many groups to either solve this or give reasonable evidence that it's not practical.

What task information to extract: This begs the question: what kind of task information is sufficient for performance portability across the array of foreseeable parallel architectures? Fortunately, the support system doesn't define this, but leaves it free to evolve as research progresses, enabling an upgrade path, and again speeding the collective search.

According to the model in part 1, three kinds of information should be sufficient: 1) manipulators, which are able to modify the size of tasks, choose among alternative binary versions, and change the data layout and access pattern (such as auto-tuners injected into the binary and invoked during installation or by plugin), 2) information about the tasks such as communication with other tasks (for data affinity), preferred core type, data footprint, and predicted execution time, and 3) real-world constraints that relate to the tasks, such as deadlines, maximum latency for data to pass from one point in the computation to another, and quality related information. Again, it is up to the language and plugin to agree on what data is extracted, passed, and then used for scheduling. Meanwhile, VMS must provide the scheduler with the services needed to make use of that data.

As research progresses, additional types of information may prove needed, so flexibility from the plugin system will be key. The plugin plus VMS implementation are a hardware abstraction – the parallel equivalent of what the instruction set used to be for sequential processors, but now defined in software. Being in software provides flexibility as research progresses.

Info exposed by VMS to plugin: This, finally, begs the question: what kind of information and services must VMS provide to the plugin? It must expose the structure of the hardware that matters most to performance (computation, energy, and real-world related performance).

As discussed in part 1, the type(s) of cores, the pools of memory and communication between them should be the most important features for parallel performance.

On the nature of communication, we believe that the scheduler can safely model any

network with a topology-independent statistical model, without undue loss of performance [7, 9]. This leaves the main feature as coherent memory vs distributed, which determines whether communication takes place by shared variables in the code vs whether it needs explicit action.

Following this, we propose that for the purposes of scheduling, most parallel hardware will be adequately modeled by a simplified hierarchical graph. Nodes are of type: 1) physical memory array, 2) processor pipeline, 3) internally-scheduled sub-graph. Communication links are chosen from: 1) automated movement (ex: due to coherence mechanism) 2) explicit movement. The links are statistical models of how well the physical network moves data.

The internally-scheduled sub-graph is the key feature. It allows an entire GPU, for example, to be treated as one run-time schedulable node as StarPU [12] does. It also allows a hierarchy of plugins to exist for complex hardware. Higher level plugins schedule large tasks to sub-graphs, in which the tasks are again divided (using manipulators packaged by the toolchain, such as demonstrated by PetaBricks and DKU[44, 40]). Techniques such as Hierarchically Tiled Arrays [39] and the loop manipulation features of the X Language [26] will also facilitate such hierarchical scheduling.

We suggest that most parallel architectures will eventually fall into a small number of groups. All targets in a group have similar graphs. This conveys sufficient structure to efficiently schedule any target in the group, without exposing chip-specific details.

Focus Area: This will be difficult research (some of which is currently in progress) developing low-overhead schedulers that have a single binary-interface, but efficiently handle a range of related hardware graphs. An example would be a single plugin whose scheduler is efficient on various multi-core systems with GPU accelerators, regardless of which particular multi-core and GPU chips are present, possibly adapting job-scheduling approaches [32].

12.2 Eco-System

Figure 12.3 depicts how many real-world entities might interact to supply the various pieces. At the top, independent software developers write applications, in a variety of languages. Each language is defined by a research group, along with its own format for conveying task-related info.

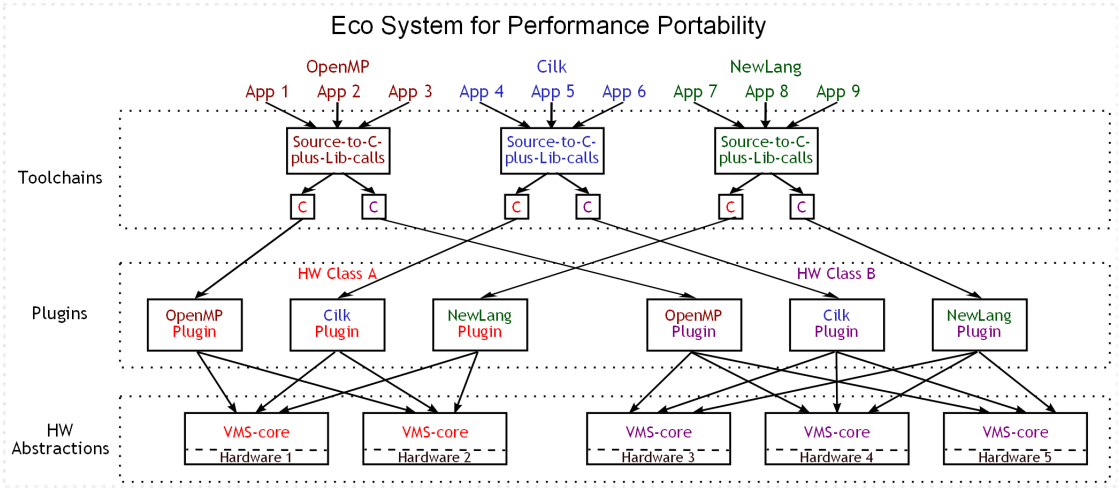

Figure 12.3: Eco-system is composed of toolchains, plugins, and HW abstractions. Each element, such as a particular plugin or sequential C compiler, is supplied by a different physical-world entity, such as a company or a research group. Elements related to a particular language are all shown in the same color, while elements related to the same hardware class are also shown in the same color. The plugins combine a language color with a hardware class color because they depend on both. As can be seen by the coloring, the toolchain for a language is independent of HW except for the sequential C compilers.

The plugins, in the middle, act as a cross-bar switch, connecting the binaries to the hardware abstractions. They are separately distributed and loaded onto the hardware, and separately written. The middle will fill out as research teams discover scheduling techniques for various groupings (classes) of hardware. Each implements a plugin for a hardware class, which accepts a particular language's format. For unusual hardware designs, the manufacturer supplies their own plugin for the popular languages, thereby taking advantage of the existing application binaries (non-standard instruction sets also need an install-time translator).

At the bottom, the VMS-core implementations standardize the hardware. They are mainly supplied by the hardware manufacturers.

The minimal software needed for a new HW platform is the VMS-core abstraction, sequential C compiler, and a bare-bones OS (and possibly a binary-optimizer). Existing applications are adapted via the plugins for the hardware class and the abstraction.

It is expected that a small number of HW classes will quickly come to dominate, which will encourage later HW development to fit within the dominant classes. As a result, a standard set of plugins, and sequential C compilers will emerge, allowing software developers to perform a single compilation pass. For fine tuning of *sequential* compilation choices (or optimization of fine-grain hardware like SIMD GPUs), we expect install-time binary re-writers or run-time binary optimizers.

The end-result is that no centralized control is needed. Language designers innovate, inventing new parallelism constructs, and need as little as just a source-to-C translator to reach all the standard hardware platforms. Likewise, hardware manufacturers innovate freely, needing minimal software development for a new chip to enjoy access to existing development tools and applications.

Note that new hardware still has the option of aggressive binary optimizers. In this scheme they have richer application information available than normal due to the task information bundled into the binary.

The set of plugins is the key to this portability, in combination with funneling many applications to the same parallelism information at the top, and funneling many hardware platforms to similar graphs at the bottom.

Chapter 13

Related Work

Other work on the combined productivity and portability challenge focuses on one aspect or another of the problem. In contrast, this dissertation's two proposals are the first for frameworks to support a collective solution emerging from individual research efforts.

After 50 years, the number of papers is in the thousands, and number of approaches in the hundreds. However, they can be classed by their general approach to the problem:

- static conversion from sequential to parallel code
- sequential code with parallelism hidden inside library functions (statically compiled for each machine)
- sequential code with parallelism hidden inside library functions (dynamically linked on machine – SEJITS)
- parallel code with constructs for specialization (statically compiled, with by-hand specialization – sequioa)
- parallel code with language forces scheduling code in application (statically compiled, incompatible with specialization – TBB, UPC.. long list)

- parallel code with language suitable for narrow class of easily-parallelised problems (statically compiled, tool-choice specialization – HPF, OpenMP, MapReduce, long list)
- parallel code with language compatible with specialization (statically compiled, tool-choice specialization – CnC, Galois)

The dimensions that classify the approaches are: 1) productivity 2) range of applications 3) specialization strength and 4) specialization process

One prominent example of a language plus framework is Sequioa. It has constructs to identify task and identify position in memory hierarchy. The programmer writes code for a particular machine that chooses where in the memory to place a given task. That splits the code into two parts: the application part that states tasks, and the scheduler-helper part that chooses placement. This makes it similar to BLISS's bidirectional library approach. However, each application has to have a scheduler-helper written just for it, for each target machine. This forces the application developer to have detailed knowledge of the hardware. Compilation is manual, and the build-developer must specify which of those "placement" modules to compile.

Within the last two categories, the specialization consists of changing the compiler to one for the target machine. This requires manual compile of the source for the target machine. This specialization is weak – the language doesn't include any features that satisfy specialization's needs. Most such languages were designed with only static compilation for a single machine in mind. However, the last category, notably CnC, were designed with specialization in mind and avoid constructs that interfere with specialization. CnC specifies constraints rather than scheduler implementations, which is a necessary ingredient for portability of source.

As discussed in part 1, a one-stop solution to performance portability fails to address critical real-world patterns. For example, attempting to place the full specialization into the toolchain centralizes the specialization effort in a single entity. One route is PetaBricks style

[10] which places specialization into a single tool that injects an adaptable runtime. BLISS also takes this approach [45], automating re-compilation for new targets and automating distribution of multiple binaries. To do this for thousands of different software development entities requires centralization. Either way, such a centralized approach concentrates control, and may create a choke-point slowing the pace of innovation.

Pure runtime based approaches [44] imply a single binary, with only the runtime dynamic-library changing across targets. This frees hardware manufacturers to independently deploy runtime libraries. However, for performance they must take advantage of language-specific constructs. This forces a separate runtime for each language. Without a means to simplify the creation of such runtimes, and reuse effort across hardware, this represents significant development effort.

Finally, hardware abstraction based approaches, such as JITs [20, 68] or VMMs [67], place the work of specializing into the virtual machine, which makes reuse difficult, forcing rewrite of JIT internals for each hardware platform, for each language. This software cost is an issue in the embedded space where new hardware is introduced often and has a limited market size to amortize the software cost. In addition, this one-stop approach requires a different JIT for each language, because it has to recognize language-specific features to specialize – or else it fails to achieve good performance. This requires extensive work, making domain-specific languages time-consuming and difficult to develop, and the multiple JITs logistically awkward.

Many recent approaches are exposing the scheduling process in one fashion or another [28, 53, 24].

Table 13.1: Table of languages vs properties relevant to productivity, portability, and adoption

	productivity	app. range	specializ. strength	specializ. process	distribution
pthread	very low	high	none	by hand	manual by target
MPI	low	high	none	by hand	manual by target
TBB	low	high	very low	link	manual by target
OpenCL	low	low	medium	static tool	manual by target
HPF	medium	low	medium	static tool	manual by target
OpenMP	medium	low	medium	static tool	manual by target
UPC	medium	medium	medium	static tool	manual by target
Sequoia	medium	medium	medium	static tool	manual by target
CNC	medium	medium	medium	static tool	manual by target
parallelising compiler	high	very low	low	static tool	manual by target
MapReduce	high	very low	medium	link	manual by target
Cilk	high	low	low	static + sched	manual by target
SEJITS	high	low	medium	dynamic	single exe
Galois	high	medium	medium	static tool	manual by target
DKU in BLISS	high	medium	high	auto	auto
WorkTable in BLISS + VMS	high	high	high	distrib + auto	auto

Chapter 14

Conclusion

The first part of this dissertation has provided background on the productivity and portability goals, which shows that it's a highly complex problem. At the base is the specialization process, which provides the portability of the source to multiple target hardware platforms. This, in turn, requires language features that support the specialization process while also being highly productive for programmers. In addition, the specialization process has to be compatible with the workflow of the software segment.

The conclusion from the background is that the problem is too complex to successfully approach in the traditional research pattern of simplifying the problem in order to show a result on one aspect. Instead, a framework is needed that supplies the integration glue. When following this framework, independent researchers are then free to simplify in accordance with the framework, which keeps the independent work coherent with other independent research, so they add to each other. The emergent result is a collective, cooperative solution to the productivity, portability, and adoptability triple challenge.

The infrastructure has to precede the language, because the language must include features that support specialization, and be compatible with the infrastructure's specialization

support. Given such an infrastructure, the search for a highly productive language can take place, while inheriting the portability support from the infrastructure, and meeting the portions of adoptability that the infrastructure provides.

The second part showed two candidates for such a framework: BLISS and VMS. After reviewing them, the conclusion is that a combination of the two is most likely to provide the best set of advantages with the fewest disadvantages for most software segments.

Finding a highly productive language needs an extensive search, with rapid and simplified experimentation. Both approaches support such a search, but VMS is the simplest and quickest to define new parallelism constructs in.

BLISS was discussed. It's an infrastructure based on a bidirectional library interface pattern plus a centralized specialization server that holds all the sources and automatically performs the first step of the specialization process on them.

BLISS encourages new language research by providing a full infrastructure that can be readily cloned. It includes a collection of sources, and a collection of specialization modules. This supplies the entire infrastructure a researcher needs, allowing them to pick one place to focus their efforts, without having to do the work to gather and/or implement the applications, toolchains, and so on.

VMS was shown to be a morphable hardware abstraction that displaces threads, replacing it at the base of the software stack. It allows both parallelism construct behavior and scheduling behavior to be plugged in. The plugin morphs the abstraction's behavior. It further allows the parallelism construct behavior to be implemented with sequential algorithms, by providing a globally consistent sequentialization of events from the time-lines in an application. It also separates the language runtime out into a separately installed module – the plugin – which is installed much the same as a device driver. The plugin encapsulates all the parts that inescapably intermix language-specific with hardware-specific. So, a plugin for each language

desired is installed on target hardware.

VMS encourages research on new languages by its sequentialization, making it exceptionally fast and simple to create the language's runtime that has new parallelism constructs. This speeds the search for highly productive language features.

Both approaches support the trend towards domain-specific languages. This is important because such domains have a relatively small number of users, so reuse of as much of the specialization, development tools, and toolchain, as possible is needed. VMS provides a clean decomposition of the specialization into three stages, which allows good reuse of the specialization effort. Its split front-end compilation into source-to-C transform plus C compiler also simplifies the creation of new languages – they only need provide the source-to-C translator, which is much simpler than a full compiler.

Bibliography

[1] Amdahl's law. http://en.wikipedia.org/wiki/Amdahl's$_law$.

[2] Damit tools for api changes. http://code.google.com/p/dam-tools/.

[3] Open Media Platform homepage. http://www.openmediaplatform.eu/.

[4] Sample BLIS Code. http://dku.sourceforge.net/SampleCode.htm.

[5] Suif parallelizing compiler homepage. http://suif.stanford.edu.

[6] Workshop on the roadmap for the revitalization of high-end computing. Available at `http://www.cra.org/reports/supercomputing.web.pdf`, jun 2003.

[7] Anant Agarwal. Limits on interconnection network performance. *Parallel and Distributed Systems, IEEE Transactions on*, 2:398 – 412, 1991.

[8] G. Agha, I. Mason, S. Smith, and C. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7(01):1–72, 1997.

[9] K. Aida, A. Takefusa, H. Nakada, S. Matsuoka, S. Sekiguchi, and U. Nagashima. Performance evaluation model for scheduling in global computing systems. *INTERNATIONAL JOURNAL OF HIGH PERFORMANCE COMPUTING APPLICATIONS*, 14:268–279, 2000.

[10] Jason Ansel, Cy Chan, Yee Lok Wong, Marek Olszewski, Qin Zhao, Alan Edelman, and Saman Amarasinghe. Petabricks: a language and compiler for algorithmic choice. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 38–49, 2009.

[11] Spiral Group at CMU. Spiral homepage. http://www.spiral.net.

[12] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009*, 23:187–198, 2011.

[13] Arnaldo Azevedo, Cor Meenderinck, Ben Juurlink, Andrei Terechko, Jan Hoogerbrugge, Mauricio Alvarez, and Alex Ramirez. Parallel h.264 decoding on an embedded multicore processor. In *HiPEAC '09: Proceedings of the 4th International Conference on High Performance Embedded Architectures and Compilers*, pages 404–418, 2009.

[14] Ron Baecker, Chris DiGiano, and Aaron Marcus. Software visualization for debugging. *Communications of the ACM*, 40(4):44–54, 1997.

[15] T. A. Ball and S. G. Eick. Software visualization in the large. *IEEE Computer*, 29(4):33–43, apr 1996.

[16] Muthu Manikandan Baskaran, Uday Bondhugula, Sriram Krishnamoorthy, J. Ramanujam, Atanas Rountev, and P. Sadayappan. A compiler framework for optimization of affine loop nests for gpgpus. In *ICS '08: Proceedings of the 22nd annual international conference on Supercomputing*, pages 225–234, 2008.

[17] G. Berry and G. Boudol. *The chemical abstract machine.* ACM Press, 1989.

[18] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H.

Randall, and Yuli Zhou. Cilk: an efficient multithreaded runtime system. *SIGPLAN Not.*, 30(8):207–216, 1995.

[19] JR Burch, EM Clarke, KL McMillan, DL Dill, and LJ Hwang. Symbolic model checking: 10^{20} states and beyond. *Logic in Computer Science, 1990. LICS'90, Proceedings*, pages 428–439, 1990.

[20] B. Catanzaro, S. Kamil, Y. Lee, K. Asanovic, J. Demmel, K. Keutzer, J. Shalf, K. Yelick, and A. Fox. Sejits: Getting productivity and performance with selective embedded jit specialization. *First Workshop on Programmable Models for Emerging Architecture at the 18th International Conference on Parallel Architectures and Compilation Techniques*, 2009.

[21] A. Church. The calculi of lambda-conversion. *Annals of Mathematics Studies*, (6), 1941.

[22] Charles Consel. A general approach for run-time specialization and its application to c. pages 145–156. ACM Press, 1996.

[23] Google Corp. MapReduce home page. http://labs.google.com/papers/mapreduce.html.

[24] Intel Corp. CnC homepage. http://software.intel.com/en-us/articles/intel-concurrent-collections-for-cc/.

[25] Intel Corp. TBB home page. http://www.threadingbuildingblocks.org.

[26] Sebastien Donadio, James Brodman, Thomas Roeder, Kamen Yotov, Denis Barthou, Albert Cohen, Mara Garzarn, David Padua, and Keshav Pingali. A language for the compact representation of multiple program versions. In *Languages and Compilers for Parallel Computing*, Lecture Notes in Computer Science, pages 136–151. 2006.

[27] E Duesterwald. Design and engineering of a dynamic binary optimizer. *Proceedings of the IEEE*, 93:436 – 448, 2005.

[28] Kayvon Fatahalian, Daniel Reiter Horn, Timothy J. Knight, Larkhoon Leem, Mike Houston, Ji Young Park, Mattan Erez, Manman Ren, Alex Aiken, William J. Dally, and Pat Hanrahan. Sequoia: programming the memory hierarchy. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 83, 2006.

[29] S. Fortune and J. Wyllie. Parallelism in random access machines. *STOC '78: Proceedings of the tenth annual ACM symposium on Theory of computing*, pages 114–118, 1978.

[30] M. P. I. Forum. *MPI: A Message-Passing Interface Standard.* 1994.

[31] Grigori Fursin, Albert Cohen, Michael OBoyle, and Olivier Temam. A practical method for quickly evaluating program optimizations. In Tom Conte, Nacho Navarro, Wen-mei Hwu, Mateo Valero, and Theo Ungerer, editors, *High Performance Embedded Architectures and Compilers*, Lecture Notes in Computer Science, pages 29–46. Springer Berlin / Heidelberg, 2005.

[32] Jrn Gehring and Friedhelm Ramme. Architecture-independent request-scheduling with tight waiting-time estimations. In *Job Scheduling Strategies for Parallel Processing*, volume 1162 of *Lecture Notes in Computer Science*, pages 65–88. 1996.

[33] D. Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(1):80–112, 1985.

[34] Leslie M. Goldschlager. A unified approach to models of synchronous parallel machines. In *STOC '78: Proceedings of the tenth annual ACM symposium on Theory of computing*, pages 89–94. ACM Press, 1978.

[35] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification.* Addison-Wesley, 1996.

[36] Kronos Group. OpenCL home page. http://www.khronos.org/opencl.

[37] Cilk group at MIT. CILK homepage. http://supertech.csail.mit.edu/cilk/.

[38] UPC group at UC Berkeley. Unified Parallel C homepage. http://upc.lbl.gov/.

[39] Jia Guo, Ganesh Bikshandi, Basilio B. Fraguela, Maria J. Garzaran, and David Padua. Programming with tiles. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, PPoPP '08, pages 111–122, 2008.

[40] Sean Halle. DKU-ized Deblocking Filter code. http://dku.svn.sourceforge.net/viewvc/dku/branches/DKU_C__Deblocking__orig/.

[41] Sean Halle. VMS home page. http://vmsexemodel.sourceforge.net.

[42] Sean Halle. A mental framework for use in creating hardware independent parallel languages, 2006. http://www.soe.ucsc.edu/share/technical-reports/2006/ucsc-crl-06-12.pdf.

[43] Sean Halle and Albert Cohen. BLIS website, November 2008. http://blisframework.org.

[44] Sean Halle and Albert Cohen. Dku pattern for performance portable parallel software, 2009. http://www.soe.ucsc.edu/share/technical-reports/2009/ucsc-soe-09-06.pdf.

[45] Sean Halle and Albert Cohen. Leveraging semantics attached to function calls to isolate applications from hardware. In *HOTPAR '10: USENIX Workshop on Hot Topics in Parallelism*, June 2010.

[46] Sean Halle, Dmitry Nadezhkin, and Albert Cohen. A framework to support research on portable high performance parallelism, 2010. http://www.soe.ucsc.edu/share/technical-reports/2010/ucsc-soe-10-02.pdf.

[47] Wilhelm Hasselbring. Programming languages and systems for prototyping concurrent applications. *ACM Comput. Surv.*, 32(1):43–79, 2000.

[48] Carl Hewitt. Actor model of computation, 2010. http://arxiv.org/abs/1008.1459.

[49] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[50] Intel. Parallel computing: Background. http://www.intel.com/pressroom/kits/upcrc/ parallel-computing_backgrounder.pdf.

[51] Ken Kennedy and John R. Allen. *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers Inc., 2002.

[52] M Kharbutli, Y Solihin, and L Jaejin. Eliminating conflict misses using prime number based cache indexing. *IEEE Transactions on Computers*, 54(5):573–586, 2005.

[53] Kathleen Knobe. Ease of use with concurrent collections (CnC). In *HOTPAR '09: USENIX Workshop on Hot Topics in Parallelism*, March 2009.

[54] C. H. Koelbel, D. Loveman, R. Schreiber, and G. Steele Jr. *High Performance Fortran Handbook*. MIT Press, 1993.

[55] Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L. Paul Chew. Optimistic parallelism requires abstractions. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 211–222, 2007.

[56] G Leavens and M Sitaraman (eds). *Foundations of Component-Based Systems*. Cambridge University Press, 2000.

[57] J McGraw, S. Skedzielewski, S. Allan, and R Odefoeft. *SISAL: Streams and Iteration in a Single-Assignment Language: Reference Manual Version 1.2*. Lawrence Livermore National Laboratory, 1985. Manual M-146 Rev. 1.

[58] J. McGraw, SK Skedzielewski, SJ Allan, RR Oldehoeft, J. Glauert, C. Kirkham, B. Noyce, and R. Thomas. SISAL: Streams and iteration in a single assignment language: Reference manual version 1.2. *Manual M-146, Rev*, 1.

[59] R. Milner. *A Calculus of Communicating Systems, volume 92 of Lecture Notes in Computer Science*. Springer-Verlag, 1980.

[60] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, parts I and II. *Information and Computation*, 100(1):1–40 and 41–77, 1992.

[61] Robin Milner. *Communicating and Mobile Systems: The pi-Calculus*. Cambridge University Press, 1999.

[62] open-mpi organization. Open MPI home page. http://www.open-mpi.org.

[63] OpenMP organization. OpenMP home page. http://www.openmp.org.

[64] David Patterson, Susan Eggers, and Mark Horowitz. Revitalizing computer science research. http://www.cra.org/uploads/documents/resources/rissues/computer.architecture.pdf, 2005. CRA: Computing Research Association report.

[65] B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[66] Louis-Noël Pouchet, Cédric Bastoul, Albert Cohen, and John Cavazos. Iterative optimization in the polyhedral model: part ii, multidimensional time. In *ACM SIGPLAN conference on Programming language design and implementation (PLDI)*, pages 90–100, 2008.

[67] M. Rosenblum and T. Garfinkel. Virtual machine monitors: current technology and future trends. *IEEE Computer*, 38:39 – 47, 2005.

[68] Ajeet Shankar, S. Subramanya Sastry, Rastislav Bodík, and James E. Smith. Runtime specialization with optimistic heap analysis. In *Proceedings of the 20th annual ACM SIGPLAN*

conference on Object-oriented programming, systems, languages, and applications, OOPSLA '05, pages 327–343, 2005.

[69] A. Turing, 1938.

http://www.turingarchive.org/intro/

http://www.turing.org.uk/sources/biblio4.html

http://web.comlab.ox.ac.uk/oucl/research/areas/ieg/e-library/sources/tp2-ie.pdf.

[70] J. von Neumann. *First Draft of a Report on the EDVAC.* United States Army Ordnance Department, 1945.

[71] Wikipedia. HPF wikipedia page. http://en.wikipedia.org/wiki/High_Performance_Fortran.

[72] G. Winskel. *The Formal Semantics of Programming Languages.* MIT Press, 1993.

[73] Xiaolan Zhang, Zheng Wang, Nicholas Gloy, J. Bradley Chen, and Michael D. Smith. System support for automatic profiling and optimization. In *Proceedings of the sixteenth ACM symposium on Operating systems principles*, SOSP '97, pages 15–26, 1997.