2012-05-25

Automatic Discovery and Exposition of Parallelism in Serial Applications for Compiler-Inserted Runtime Adaptation

David A. Greenland
Brigham Young University - Provo

Automatic Discovery and Exposition of Parallelism in Serial Applications for Compiler-Inserted Runtime Adaptation

David Greenland

A thesis submitted to the faculty of
Brigham Young University
in partial fulfillment of the requirements for the degree of

Master of Science

David A. Penry, Chair
Brad L. Hutchings
Brent E. Nelson

Department of Electrical and Computer Engineering

Brigham Young University

June 2012

ABSTRACT

Automatic Discovery and Exposition of Parallelism in Serial
Applications for Compiler-Inserted Runtime Adaptation

David Greenland
Department of Electrical and Computer Engineering
Master of Science

Compiler-Inserted Runtime Adaptation (CIRA) is a compilation and runtime adaptation strategy which has great potential for increasing performance in multicore systems. In this strategy, the compiler inserts directives into the application which will adapt the application at runtime. Its ability to overcome the obstacles of architectural and environmental diversity coupled with its flexibility to work with many programming languages and styles of applications make it a very powerful tool. However, it is not complete. In fact, there are many pieces still needed to accomplish these lofty goals.

This work describes the automatic discovery of parallelism inherent in an application and the generation of an intermediate representation to expose that parallelism. This work shows on six benchmark applications that a significant amount of parallelism which was not initially apparent can be automatically discovered. This work also shows that the parallelism can then be exposed in a representation which is also automatically generated. This is accomplished by a series of analysis and transformation passes with only minimal programmer-inserted directives. This series of passes forms a necessary part of the CIRA toolchain called the concurrency compiler. This concurrency compiler proves that a representation with exposed parallelism and locality can be generated by a compiler. It also lays the groundwork for future, more powerful concurrency compilers.

This work also describes the extension of the intermediate representation to support hierarchy, a prerequisite characteristic to the creation of the concurrency compiler. This extension makes it capable of representing many more applications in a much more effective way. This extension to support hierarchy allows much more of the parallelism discovered by the concurrency compiler to be stored in the representation.

Keywords: Compiler-Inserted Runtime Adaptation, CIRA, concurrency compiler, hierarchy, task descriptor, relationship descriptor

ACKNOWLEDGMENTS

The effort needed to complete this thesis was more than expected and I often found myself discouraged at my lack of progress. I constantly found my spirits lifted by many individuals. I would like to recognize my advisor Dr. David A. Penry who pushed me to continue my efforts and helped me work through difficult problems. I would also like to recognize my other university professors who instilled in me a genuine interest in electrical and computer engineering. I would like to thank my parents, Richard and Silvia, who always believed in me and showed interest in my work. I would like to also thank my in-laws, David and Nancy, who often asked how this thesis was coming along and never doubted that I would finish it. Most importantly I need to thank my wife, Katie, for all of her support. She went through all of the ups and downs with me and never complained. She always had more confidence in my ability to accomplish this than I did. I could not ask for better support.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

For many years, computer performance has risen steadily. These performance improvements have become so commonplace that consumers, as well as developers, have come to expect newer computers to significantly outperform older ones. This performance increase has mostly been due to the steady increase in processor clock frequency. Unfortunately, this steady increase in clock frequency has slowed considerably causing developers to search for other methods to increase performance.

One popular method to increase performance without increasing clock frequency is to put multiple processing cores on a single chip. Unfortunately, as more and more cores are placed on a single chip, performance has not risen as expected. Some of the challenges to achieving better performance from multicore processors are the factors in play that are unknown during application development [1]. These factors include architectural details of the system, operating environment of the system, and parallelism which may not be apparent due to irregular data access patterns [2].

Perhaps the most evident solution to these challenges would be to change the style of programming. This would include changing programming languages. Many new parallel languages and models as well as extensions to existing languages have been developed [3, 4, 5, 6, 7, 8]. Because few programmers know how to use these new languages, a second piece to this solution is to teach programmers how to use them. To be proficient in these new languages, programmers would not only have to learn the new constructs and syntax of a new language but also learn to think in a parallel way. They would have to think about data dependences and determine which pieces of their program can be run in parallel. This is an enormous task.

Instead, perhaps a better solution would be to take a sequential program, or perhaps a mostly sequential program that only has a few extra directives inserted into it, and then parallelize it. This solution would remove the burden of determining parallelism from the programmers and

places it on the parallelization tools. The problem now becomes developing tools that can effectively parallelize sequential programs.

1.1 Compiler-Inserted Runtime Adaptation (CIRA)

Effective parallelization tools need to handle the issues with diversity that plague the multicore world. There are two classes of diversity that must be addressed:

Architectural diversity Multicore processors increase the dimensions along which processor and system architectures can vary. The obvious dimensions will include the number of cores, the complexity of cores, memory hierarchies, and presence of specialized cores. Additional dimensions include the presence or absence of specialized architecture features such as transactional memory or inter-core communication queues. Heterogeneity of cores will also be very likely [9]. Heterogeneity will be further caused by increasing transistor variability as feature sizes shrink; different cores with the same design will attain to different frequencies on the same die.

Environmental diversity In the past, significantly parallel applications have typically run in an environment in which the resources available, such as the number of cores, were fixed at compile time or at program initialization. As a result, programmers have generally worried about variation in resources only at a very coarse-grained level and only for some applications. However, more widespread use of parallelism in multicore systems will result in multiple parallel applications running simultaneously; the operating system will wish to allocate resources dynamically between applications as system load changes. Thus there will be much run-to-run diversity in the environment as well as within-run diversity.

The key to solving the diversity problem is adaptation. The application must adapt to the architecture and environment in order to run most efficiently. Unfortunately, this adaptation cannot be carried out at development or compile time because neither the architecture nor the environment are known at those times. Applications will be run on more than just one architecture. Even if the application could be adapted to the system it is running on, different runs at different times will have different resources available because of the environmental diversity. The application not only has to be adapted to the architecture it will run on, it must also be adapted at runtime

to the resources available. Developing an adaptive application greatly increases the difficulty of writing a parallel application by forcing the programmer to deal with deep performance issues of cache hierarchy, locality, load balancing, and concurrency. Compiler-Inserted Runtime Adaptation (CIRA) seeks to free programmers from the need to concern themselves with these performance issues.

CIRA is a philosophy that addresses both types of diversity. How the application needs to be adapted is not known at compile time, but the steps needed at runtime to determine how to adapt the application can be known at compile time. The CIRA philosophy uses adaptation directives specifying what decisions need to be made at runtime and how to make those decisions in order to adapt most effectively. At runtime these directives are able to adapt the program according to the architecture and the environment. Hence the name Compiler-Inserted Runtime Adaptation. This philosophy will free the programmers from worrying about diversity and allow them to focus on creating correct and efficient programs. It also allows a wide range of programs to use multicore processors to achieve performance improvement.

CIRA works by converting the existing program into a representation that exposes the parallelism inherent in the program. This is called an exposed parallelism and locality (EPL) representation [10, 11]. This step is accomplished by a concurrency compiler. This EPL representation of the program is then adapted to the system it is running on to further take advantage of available parallelism. This step is accomplished by an adaptation compiler. CIRA, including the EPL representation, will be described in detail in chapter 3.

1.2 Contributions

1.2.1 Concurrency Compiler

The very first step in the CIRA toolchain is the conversion of the application into the EPL representation. Every other step in the toolchain relies on a correct EPL representation of the application, so this step must happen first. This conversion must be accomplished with minimal help from the programmer. As concurrency compilers are developed to support more styles of programs, CIRA becomes a more powerful tool. Concurrency compilers can also be optimized to make a more efficient EPL representation. Before any of these optimizations, extensions, and

additions can be developed, we must show that the EPL representation can be generated automatically.

The main contribution of this work is demonstrating the automatic discovery of the parallelism in an application and the generation of the EPL representation to expose that parallelism. This is demonstrated by the creation of the first concurrency compiler. This first concurrency compiler includes analyses that discover the parallelism inherent in an application and transforms that convert it into the EPL representation. Though this is a rather simple concurrency compiler and will eventually need optimizations, extensions, and additions as mentioned previously, it shows that the concept works. It shows that parallelism in a serial application can be automatically discovered and exposed with only minimal help from the programmer. Many of these analyses and transforms developed for this first concurrency compiler will be necessary for all future concurrency compilers. Therefore, this work is not only a proof of concept, but it also lays the groundwork for all future concurrency compilers.

1.2.2 Hierarchy

A secondary contribution of this work is the extension of the EPL representation to support hierarchy. The inability to support hierarchy severely restricted the set of applications the EPL representation could represent effectively. With hierarchy support, the EPL representation is very powerful and can support most applications in a much more effective way. Hierarchy allows much more parallelism and locality to be expressed. It also allows more of the parallelism discovered by the concurrency compiler to be effectively stored. Extending the EPL representation to support hierarchy was a needed prerequisite to developing a concurrency compiler.

1.3 Outline

The remainder of this thesis is organized as follows. Chapter 2 presents work done on discovering parallelism in sequential applications. It covers many parallel programming languages and models. Chapter 3 describes background information needed to understand this work. This includes more details on each step in the CIRA toolchain as well as a detailed description of the EPL representation. Chapter 4 describes the prerequisite contribution of extending the EPL representation to support hierarchy. Chapter 5 contains the details of developing the first concurrency

compiler proving that parallelism inherent in an application can be automatically discovered and exposed. This includes a description of analyses and transforms used in this concurrency compiler which lay a foundation for the development of additional concurrency compiler in the future. Chapter 6 presents experimental data for six benchmark programs. In addition to showing that the programs run successfully through the CIRA toolchain, the data also shows code size of the generated EPL representation, concurrency compilation time, and other metrics that can all be improved through optimization of future concurrency compilers. Chapter 7 summarizes the accomplishments of this thesis.

Chapter 2

Related Work

The increasing need to parallelize sequential applications has prompted research in automatic discovery of parallelism. This chapter briefly describes some of the more prevalent methods. Understanding other methods of automatic discovery of parallelism will aid in understanding the unique strengths of this work.

2.1 Compile Time Techniques

Many parallel programming languages have demonstrated how useful programmer-inserted high-level language constructs can be during discovery. For example, a DOALL construct typically implies that the programmer has guaranteed that the loop iterations can run in parallel on all possible executions of the program. Parallel programming languages can further simplify discovery by using constructs which explicitly state concurrency and locality information. Some of the more common parallel programming languages and models include [3, 4, 5, 6, 7, 8]. Due to the large amount of information available in high-level constructs most of the previous work on automatic discovery of parallelism uses these type of constructs and focuses on either loops and arrays [12, 13] or control-flow structures [14].

Other methods used in discovery of parallelism include using a message-driven programming model. The CHARM++ [8] programming language uses this programming model and includes a notion of actors as well as task priorities and dynamic scheduling. CHARM++ attempts to load balance without considering locality. Another approach is to use a library to aid in discovery of parallelism.

Another method is to develop an extension to a sequential programming language. pC++ [6, 7] is an object-parallel extension to C++ which provides the user with data structures that are distributed across multiple processors. These data structures are called *distributed aggregates*.

Many compiler tools have also been developed to work on specific parallel programming languages, programming models, or extensions to sequential programming languages. These compiler tools generally use analysis of data accesses. The often focus on parallelizing loops or constructing a task-based representation of the application.

The ParaScope compiler tool suite has a special representation for inter-procedural side-effect analysis using Regular Section Descriptors (RSDs) [15]. Coarse-granularity parallelism can be found by determining that code sections are independent of others, especially code that operates on an array. The compiler separates code sections for execution when it can prove that different code blocks or iterations of a loop operate on independent sections of an array. ParaScope works on FORTRAN input and creates parallelism by transforming sequential DO loops into parallel DO loops.

ParaScope approaches proving independence through regular sections: an analysis of the access patterns across an array. A *regular section* is a commonly used access pattern over a portion (*section*) of an array, such as a column or row. A regular section defines a set of actual data accesses. Two refinements of the RSD are the Data Access Descriptor (DAD) [16] and the Processor Tagged Descriptor (PTD) [17]. The main contribution of the DAD is the *simple section* which is a simple boundary around the accessed portion of an array. The PTD more precisely defines that bounded area of the array which allows for more complex shapes for describing irregular accesses.

A different approach is used in Pilar's internal representation, Communication Pattern Internal Representation (CPIR) [18, 19]. The Pilar compiler uses three basic constructs to represent data access patterns: *intervals*, *enumerations* and *cyclics*. Intervals, like the descriptors previously described, are a set of bounds on the index ranges for data access. Enumerations allow for completely irregular access patterns and are simply a list of all individual accesses. Cyclics provide a specific access pattern and allow for a smaller memory footprint and faster set operations.

An access schedule, which is the ordering of element access within the construct, is also created in addition to the data pattern. This schedule is then used to better analyze the relationships between different patterns. Since CPIR is a message-passing system, the schedule consists of the sequence of messages to communicate CPIR primitives, which are then translated to the actual local addresses. Pilar has the ability to describe very regular and irregular systems and some

capability to describe simple patterns when necessary. The pattern is limited to simple, strided array accesses, which results in lower overhead for those applications that it describes well.

The Polaris compiler's Internal Representation (IR) [20] has a 1-1 relationship with basic FORTRAN constructs using a high-level format. Expressions, statements, and symbol tables are basic classes. Parallel loops, after appropriate analysis, become doall loops, where each iteration may be executed in parallel. Variables may be declared as private to the loop.

Polaris is similar to the OpenMP [21] standard. OpenMP is a set of compiler directives for C and FORTRAN programs, including automatic partitioning of loops and parallel sections to processors, synchronization through barriers, etc. The parallelism described by OpenMP is explicit, specifying which variables are private and which loops may be executed in parallel.

The Stanford University Intermediate Format (SUIF) compiler [22, 23] is an extensive project to increase the ability of compilers to automatically extract parallelism from existing benchmarks. Internally, programs are represented on a relatively high level using loops, conditional statements, and array accesses, as well as the more common low-level information used by a compiler. High-level information is in a canonical form for the compiler passes to use. After the high-level passes have completed, the high-level information is transformed into lower-level instructions.

2.2 Runtime Techniques

A major limitation of many of the current approaches is that they require all discovery to take place at compile time. If the compiler cannot prove that all possible runs of the application will exhibit concurrency, it must conclude that the work cannot be performed in parallel. This limitation could cause much of the parallelism to be lost. Even runtime loop-level parallelization techniques such as [24, 25] have this limitation.

One exception to this limitation is the inspector-executor model of parallel execution [26]. This model uses a two-stage method to discover parallelism and run the application using the parallelism discovered. The first stage, the inspector, examines the application's data dependences between loop iterations at runtime. The second stage, the executor, then schedules loop iterations based upon those data dependences. The use of a runtime inspector relieves the compiler from having to prove everything about concurrency at compile time.

Other methods useful in discovering some parallelism at runtime are called speculative parallelization techniques [27]. These techniques discover parallelism at runtime for the sole purpose of detecting mis-speculation. One such technique is called speculative thread-level parallelism. In instances where parallelism cannot be proved at compile time, it speculatively assumes the work can be performed in parallel. A correct state must be stored for recovery when the assumption is incorrect. It then discovers at runtime if the assumed parallelism actually exists and either continues or recovers to the correct state.

KELP [4, 5] is a C++ library designed to simplify the specification of data-parallel operations on irregular block-wise data layouts for message-passing architectures. KELP succeeds in representing regular applications and some irregular applications, although irregularity must be described in blocks. Some very irregular programs see a high overhead from the block-wise decomposition. In addition, KELP is a strictly runtime-only environment, eliminating compile-time analysis of the representation.

Another runtime-only method for discovering and exposing parallelism was proposed by Johnson in [28] and uses a task graph representation. The tasks are assigned states which specify whether or not they are ready to run, currently running, or finished. As tasks become available they are processed by threads. A task structure handles dynamic creation and execution of tasks. Use of this task graph requires good heuristics appropriate to the problem in order to properly and efficiently execute tasks. The nature of this task graph makes both regular and irregular applications easy to represent. Communication costs and dependencies are represented as part of the task graph. Because the tasks are dynamically assigned at runtime, the execution overhead can be very large, especially for fine-grained applications.

The Hierarchical Task Graph (HTG) [29] was made as part of the autoscheduling project headed by Polychronopoulos. The HTG describes a task graph in a hierarchical manner by combining loops into a single node. These nodes may be further collapsed to change the granularity of the task graph. The edges in the graph represent the communication cost and dependencies between tasks. The overall preconditions necessary to execute a task are represented by a set of *execution tags*. Redundant tags may be optimized for run-time efficiency. While the HTG allows for some static analysis at compile time, its focus is on the dynamic runtime scheduling of tasks.

2.3 Summary

Many languages and tools have been developed to discover and expose parallelism in applications. The majority of these focus only on compile time analysis. Much of the parallelism in many applications cannot be proven at compile time, so these methods cannot discover a large amount of possible parallelism. Other methods focus only on runtime discovery which introduces a large amount of overhead during execution. Only a handful of tools use both static compile time analysis and dynamic runtime analysis to both discover more parallelism and keep execution overhead low. This work expands on these methods.

Chapter 3

Background

This chapter presents the background information necessary to understand the contributions of this work. This information includes the CIRA philosophy and the LLVM compiler framework which is used extensively in this work.

3.1 Compiler-Inserted Runtime Adaptation (CIRA)

The basic concepts that make up CIRA were originally introduced by Penry in [10], and the entire CIRA philosophy was later presented in his CAREER proposal to the National Science Foundation. The goal of CIRA is to free programmers from having to deal with architectural and environmental diversity. This goal is accomplished through pioneering new compilation and runtime adaptation strategies in which the compiler adds adaptation to an application. This approach enables software developers to ship parallel applications which provide excellent performance to the end user, despite the presence of diversity. As a result, multicore processor systems will be able to meet users' increased performance expectations and will justify continued development of further generations of microprocessors.

Figure 3.1 provides a block diagram of a CIRA-based application development and deployment flow. The flow begins with the developer creating an application in a programming language and model of his or her choice. It is unlikely that a single programming model or language will prove to be compellingly superior to all others; therefore, CIRA will include support for a wide variety of languages supporting a wide variety of programming models, from general-purpose languages to data-parallel languages to domain-specific languages.

Next, a *concurrency compiler* translates from the high-level language to a common intermediate representation of the application called an *exposed parallelism and locality* (abbreviated

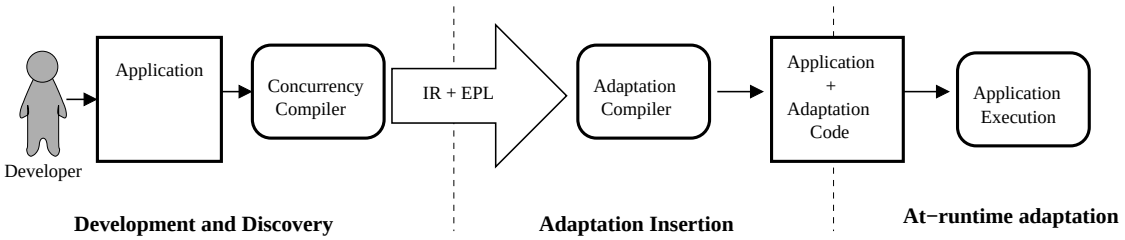

Figure 3.1: Compiler-Inserted Runtime Adaptation (CIRA)

as *EPL*) representation. The roles and challenges for the concurrency compiler are described in section 3.1.1. The EPL representation will be described in detail in section 3.1.2.

Next, an *adaptation compiler* analyzes the EPL representation, chooses an adaptation strategy to use, and inserts code to perform runtime adaptation into the application. The roles and challenges for the adaptation compiler are described in section 3.1.3.

Finally, at execution time the application adapts to the architecture and environment. This adaptation takes into account the number of available cores, the mix of cores, cache hierarchy, and on-chip communication networks.

3.1.1 Concurrency Compiler

The concurrency compiler translates from the high-level language to the EPL representation. One of the main difficulties in exploiting parallelism in applications is that the parallelism is not evident. Some programming languages make it easier by using annotations or special types describing the parallelism [6,8,21,30,31], but most do not. The parallelism needs to be discovered. A concurrency compiler discovers the parallelism in an application and makes that parallelism evident by transforming the application into an intermediate representation which explicitly shows the parallelism.

Discovery of parallelism and locality involves determining the units of work (tasks) in an application that could potentially be executed in parallel, the runtime conditions under which they actually can be executed in parallel, additional ordering relationships between tasks that cannot be executed in parallel, and what data is shared between the tasks. Tasks can have various granularities; function invocations, loop iterations, or even finer-granularity elements could be potential tasks. Because the EPL representation is a form of task graph consisting of task descriptors and re-

lationship descriptors, which can be instantiated into tasks and relationships, the basic function of the concurrency compiler's analyses is determining how to break the code up into task descriptors and relationship descriptors.

Eventually there could be a concurrency compiler for each supported programming language. Each concurrency compiler will be able to find any user annotations describing parallelism as well as perform analyses on the application to discover more parallelism. User annotation will be different or even non-existent depending on which programming language the application is written in. The analyses may need to be different depending on the programming language. Though the concepts employed by the analyses may be very similar, the differences in the language semantics and structure may necessitate separate concurrency compilers.

The main contribution of this work is the development of the first concurrency compiler. This concurrency compiler is for the C and C++ programming languages. The only user annotations required are to state which sections of code should be parallelized. This concurrency compiler is designed to be very flexible with these annotations so that an application could even just have annotations specifying that the whole application needs to be parallelized. Since there is a significant amount of overhead to the parallelization, a more practical method for annotating an application would be to determine which parts of the code would improve performance most by parallelization and just place the annotations around those parts. Many analyses in determining task descriptors, task instantiation functions, relationship descriptors, and relationship instantiation functions have been developed. These analyses lay the foundation for creating more effective, efficient, and complete analyses in the future. They also lay the foundation for creating similar analyses for other programming languages. This contribution is described in detail in chapter 5.

3.1.2 Exposed Parallelism and Locality (EPL) Representation

The parallelism that the concurrency compiler discovers must be available for the adaptation compiler to use. This necessitates an intermediate representation for that parallelism. It is important that this representation includes all parallelism and locality information discovered by the concurrency compiler and that this information is exposed for the adaptation compiler to use it. This representation should be easily generated from common programming models. It should also be capable of representing not just parallelism and locality which can be proven at compile time,

```
for (int i = 2 ; i < 8; ++i) {
 A[i] = A[i-2] + 1;
}
```

Figure 3.2: A simple loop.

but also the runtime conditions under which parallelism or locality *might* exist. We call this representation an exposed parallelism and locality (EPL) representation which was originally presented in [11] and [10].

The EPL representation is a connecting piece in the CIRA flow. Many concurrency compilers could be developed for many different programming languages. There will also be many adaptation strategies developed for the adaptation compiler. The EPL representation is generic to programming language and adaptation strategy. It can represent any programming language a concurrency compiler is developed for. This trait means the adaptation compiler never needs to change based on which concurrency compiler was used.

Both of the contributions of this work are very closely linked with this representation. The main contribution, the development of the first concurrency compiler, transforms an application into the EPL representation. The secondary contribution, the extension of the EPL representation to support hierarchy, is a direct extension of this representation. To understand the details of each of these contributions, the EPL representation must be described in detail.

The EPL representation is a form of task graph but is more powerful. A task graph is a way to represent an application as a collection of tasks and relationships. In a task graph the nodes in the graph represent the tasks. Tasks can be large pieces of the application including entire functions or even more, or they can be much smaller, perhaps only a few instructions. The edges in the graph represent the relationships between the tasks. An edge between two tasks would mean that there exists a relationship between the tasks. Often this could be a dependence relationship stating that one task must be executed before the other task. Ordering, exclusive, sharing, and nearness relationships further specify task ordering to improve performance. Tasks can be scheduled to be executed in a correct and efficient order by observing the relationships in the graph. To aid in understanding the EPL representation we will use a simple loop, shown in figure 3.2 as an example. Figure 3.3 shows the task graph for the loop.

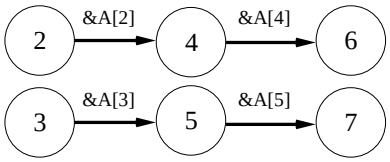

Figure 3.3: The task graph for the simple loop.

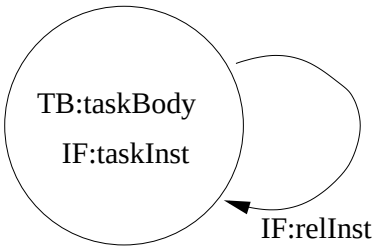

Figure 3.4: The task descriptor graph for the simple loop.

The EPL representation can more efficiently depict the application structure than a task graph because of its use of task and relationship descriptors to form a task descriptor graph. Task descriptors describe a similar group of tasks. Similarly, relationship descriptors describe a similar group of relationships. Using descriptors makes the representation much more concise. When needed, the task and relationship descriptors can still be expanded into individual tasks and relationships called instances. This expansion is called instantiation. We anticipate that for many adaptation strategies, the tasks will not need to be instantiated. The descriptors themselves will hold sufficient information, so instantiating the individual instances would only add overhead. A task descriptor graph is a graph with task descriptors as nodes and relationship descriptors as edges. Figure 3.4 shows the task descriptor graph for the loop example.

Functional Representation Another powerful characteristic of the EPL representation is that it is a functional representation. A functional representation is powerful for many reasons.

- It can describe any possible pattern of relationships. Functions are completely general, and therefore, have no restrictions on what they can describe.
- A functional representation is not static. It changes with parameters that are passed to the functions. This trait is necessary for runtime adaptation.

```
taskBody(int *&A, int i) {
  A[i] = A[i-2] + 1;
}
```

Figure 3.5: The task body function.

```
taskInst(int *&A) {
  for (int i = 2 ; i < 8; ++i){
    int name[1] = {i};
    createtask(name, COST, i);
  }
}
```

Figure 3.6: The task instantiation function.

- Functions are easily generated from a program which is already made of functions. Code from the program can easily be pulled into functions that make up the representation.
- Functions give exact results. Decisions regarding adaptation are much easier when based on exact results.
- Functions can be optimized. Compilers have many optimization strategies that are made to run on functions.

All pieces of the representation are either functions or calls to functions. For example, task descriptors are represented as calls to the TASKDESC function. Each task descriptor points to a task body function which is the code the task must run. The task body function for the previous example is shown in figure 3.5. The task descriptors also each point to a task instantiation function, shown in figure 3.6, which is the code for instantiating the individual task instances. Similarly relationship descriptors are represented as a call to the RELDESC function. Each relationship descriptor points to a relationship instantiation function, shown in figure 3.7, which is the code for instantiating the individual relationship instances. Each of these calls and functions will be described in more detail later.

Using functions to describe the task descriptors allows us to pass information via arguments to the function. Information passed to the functions varies in scope. Varying levels of scope helps

```
relInst(int *&A, int i1, int i2) {
  if (&A[i1] == &A[i2-2]){
    int name1[1] = {i1};
    int name2[1] = {i2};
    createrel(name1, 0, name2, 0, RAW, oneToTwo, NULL, COST);
  }
}
```

Figure 3.7: The relationship instantiation function.

organize the information and also increases performance by keeping information closer to where it will be needed. The variables holding information can be categorized into four types based on scope.

- *Descriptor time variables* hold information that is defined at the time the task descriptor is created. These variables exist in the context of the task descriptor. All task instances within the task descriptor have access to these variables. Descriptor time variables are passed into the task instantiation and task body functions. Of the four types of variables defined in the EPL representation, descriptor time variables have the widest scope.

- *Parent variables* hold information that is defined when a parent task instance is instantiated. These variables are passed down to all subtask instances but no other task instances have access to these variables. Parent variables have a more narrow scope. They are a part of the extension for hierarchy and will be described in detail in chapter 4.

- *Instantiation time variables* hold information that is defined when a task instance is instantiated. They are passed into the body function. These variables are not passed down to any subtask instances. They only exist within the context of the task instance. Instantiation time variables have an even more narrow scope.

- *In and out variables* are also defined within the context of a task instance. They are used to specify information that is passed along relationships. These variables are the most narrow in scope. Though the EPL representation works without these variables, they can drastically improve performance by localizing information. By passing information along a relationship

via in and out variables, the representation can avoid continually accessing descriptor time variables which may no longer be stored in the cache. This can avoid costly cache misses.

A functional representation is very powerful because it can represent any pattern of task or relationship instantiation. It can represent potential relationships with the exact code that will resolve to form them at runtime. This ability is very important because not all tasks or relationships can always be known at compile time. For example, a loop may have a variable number of iterations. If each iteration is a task, we cannot know how many there will be until we know the value of that variable at runtime. We also cannot know which iterations, or tasks, have relationships between them until we know the details of each iteration at runtime. The task instantiation function contains the code to determine the correct number of tasks at runtime. Similarly, the relationship instantiation function contains the code that, when executed at runtime, determines the relationships.

A final advantage of the functional representation is the ease of optimizing as well as analyzing the tasks and relationships contained in the representation. In a world driven by performance, optimization is of utmost importance. The ability to analyze is perhaps more important in CIRA. The adaptation compiler needs to determine which adaptation strategy to use. The correct adaptation strategy can only be determined by analyzing the representation.

Multi-relational Representation The EPL representation is also multi-relational, meaning it supports many different kinds of relationships. The most evident kind of relationship is a data dependence relationship. All forms of data dependences are supported including true data dependences, anti-dependences, and output dependences. The EPL representation also supports exclusive relationships which specify that the tasks cannot be run concurrently but can be run in any order. Specific ordering relationships are also part of the EPL representation.

Another very important relationship the EPL representation supports is the locality, or data-nearness, relationship. With the emergence of multicore processors, reducing cache misses has become a major concern. Relationships describing which tasks access nearby memory locations can help reduce cache misses and improve performance.

```
void parallel_section(int* A){
  static task_token t1;
  taskdesc(t1, NULL, 0, LOOP, taskInst, taskBody, 0, 1, 1, 1, A);
  reldesc(t1, t1, relInst);
}
```

Figure 3.8: The parallel section function.

Implementation The EPL representation is implemented as a set of functions for the reasons mentioned previously. These functions, called parallel section functions, contain calls to the TASKDESC and RELDESC functions. The TASKDESC and RELDESC functions do not need function bodies. The function calls contain all the needed information to define the task descriptors and relationship descriptors including pointers to the instantiation and body functions. Figure 3.8 shows the parallel section function for the example in the previous figures.

The operands to the call to TASKDESC, which contain all needed information for the task descriptor, are described in table 3.1. The first operand is an id that is unique for each task descriptor. The second operand is the id of the parent task descriptor or null if there is no parent. The third operand is the hierarchical depth. The parent and depth operands are part of the hierarchy extension and are described in more detail in chapter 4. The fourth operand is the kind of task such as loop or block. The fifth operand is a pointer to the task instantiation function. The sixth operand is a pointer to the task body function. The next four operands are the numbers of parent, instantiation time, in, and out variables an instance of the task descriptor would have. The final operands, of which there can be a variable amount, are the descriptor time variables.

The call to RELDESC is much simpler. The operands to this call are described in table 3.2. The first operand is the id for the first task descriptor. The second operand is the id for the second task descriptor. This relationship descriptor describes relationships between tasks from those two task descriptors. The last operand is a pointer to the relationship instantiation function. The relationship instantiation function has access to the descriptor time variables from each task descriptor, and therefore, does not need any variables from the relationship descriptor. For this reason, no additional variables are passed to the call to RELDESC.

The task body and task instantiation functions must also be defined in the EPL representation. The task body function for a task descriptor is the code that must be executed for each

Table 3.1: Operands to the TASKDESC call.

Operand	Description
ID	Unique identifier for the task descriptor
Parent ID	ID of the parent task descriptor
Depth	Level of hierarchy
Type	Type of task (loop, block, etc.)
Instantiation Function	Pointer to the instantiation function
Body Function	Pointer to the body function
Parent Variables	The number of parent variables, not the actual variables
Instantiation Time Variables	The number of instantiation time variables, not the actual variables
In Variables	The number of in variables, not the actual variables
Out Variables	The number of out variables, not the actual variables
Descriptor Time Variables	The actual descriptor time variables

Table 3.2: Operands to the RELDESC call.

Operand	Description
ID1	Unique identifier for the first task descriptor
ID2	Unique identifier for the second task descriptor
Instantiation Function	Pointer to the instantiation function

task that gets instantiated. The arguments to the task body function, described in table 3.3, are the descriptor time variables, parent variables, and instantiation time variables. The task instantiation function describes how the tasks are instantiated. The arguments to the task instantiation function, described in table 3.4, are the name vector, parent variables and descriptor time variables. The name vector and parent variables are part of the hierarchy extension and will be described in detail in chapter 4. Tasks are instantiated by calling CREATETASK. The call to CREATETASK, described in table 3.5, has as operands the task name, the cost associated with the task, and the list of parent and instantiation time variables that are passed to the body function.

Table 3.3: Arguments to the task body function.

Argument	Description
Descriptor Time Variables	The actual descriptor time variables
Parent Variables	The actual parent variables
Instantiation Time Variables	The actual instantiation time variables

Table 3.4: Arguments to the task instantiation function.

Argument	Description
Name Vector	The name vector received from the parent task
Parent Variables	The variables received from the parent task
Descriptor Time Variables	The actual descriptor time variables

Table 3.5: Operands to the CREATETASK call.

Operand	Description
Name	The name vector which was just created for this task instance
Cost	The cost associated with this task
Parent Variables	The actual parent variables
Instantiation Time Variables	The actual instantiation time variables

The last function defined in the EPL representation is the relationship instantiation function. The arguments to this function, described in table 3.6, are an instance iterator and the descriptor time variables for each task descriptor. This function describes how the relationships are instantiated. The relationships are instantiated by calling CREATEREL. The call to CREATEREL, described in table 3.7, has as operands the name and port number for the first instance, the name and port number for the second instance, the type of relationship, the direction of the relationship, a data pointer if necessary, and the communication cost.

3.1.3 Adaptation Compiler

The adaptation compiler analyzes the EPL representation, chooses an adaptation strategy to use, and inserts code to perform runtime adaptation into the application. Adaptation strategies could include inspector-executor [26], work-stealing [32], iteration space tiling [33] and others.

Table 3.6: Arguments to the relationship instantiation function.

Argument	Description
Instance Iterator 1	An iterator to the list of instances of the first task descriptor
Descriptor Time Variables 1	The actual descriptor time variables for the first task descriptor
Instance Iterator 2	An iterator to the list of instances of the second task descriptor
Descriptor Time Variables 2	The actual descriptor time variables for the second task descriptor

Table 3.7: Operands to the CREATEREL call.

Operand	Description
Name 1	The name vector for the first task instance
Port 1	The index of the in or out parameter for the first instance used for passing data along this relationship
Name 2	The name vector for the second task instance
Port 2	The index of the in or out parameter for the second instance used for passing data along this relationship
Type	The type of relationship such as RAW, Exclusive, Locality, etc.
Direction	The direction of the relationship, either 1 to 2 or 2 to 1
Data Pointer	A pointer to data needed for this relationship. This is seldom used.
Communication Cost	The Communication cost for this relationship

The adaptation code is specialized to the application in order to reduce its runtime overhead. A common IR and EPL representation allows there to be a single adaptation compiler.

The adaptation compiler must select an adaptation strategy to insert into the application. It may also choose to insert techniques which improve the amount of exploitable parallelism, such as thread-level speculation [34, 35, 36, 37] and transactional memory [38]. This choice should be based upon the application's characteristics as expressed in the EPL representation.

An adaptation strategy attempts to map the application's work onto computing resources in such a way as to maximize performance, minimize power, or optimize some other metric. For some applications and architectures this adaptation may be relatively simple and require a simple strategy. For example, DOALL loop nests with regular accesses to arrays on a chip multiprocessor with homogenous cores can be handled with iteration space tiling methods taking into account cache hierarchy and core availability [39, 40]. Other applications with more complex structures may require more sophisticated adaptation techniques.

There is flexibility in the time at which the choice of adaptation strategy is made. The earlier it is done, the more overhead at runtime can be saved. However, the later it is done, the more information about the architecture and environment is available to make a good choice. The best time to choose may very well depend upon the application's characteristics.

Two additional challenges the adaptation compiler must address are the development of better adaptation strategies and the task of keeping the runtime adaptation overhead small. Adaptation strategies must be able to adapt to the number of available cores, mix of cores, cache hierar-

chy, and on-chip communication networks. Prior work on runtime adaptation strategies has been able to adapt to a subset of these features. Adaptation to the number of cores is fairly common, e.g. [21, 24, 25, 39, 41, 42]. Adaptation to cache hierarchies for DOALL loops has also been studied [24,25]. Core mix has been addressed in a limited form by libraries which choose which library implementations to use based upon what hardware is available, e.g. whether a GPU is present. Except for adaptation to the number of cores, runtime adaptation strategies have not been applied to techniques used for enabling more application parallelism such as thread-level speculation or transactional memory.

The overhead of adapting the application at runtime must be small or the benefits of adaptation will be lost. Previously, reduction in runtime overhead has been attempted in two ways: through granularity adjustment and through specialization of the adaptation code. Loop splitting and loop fusion can be seen as a form of granularity adjustment. Auto-scheduling [43] used both granularity adjustment of a hierarchical task graph as well as specialization of dynamic task scheduling code. Inspector-executor adaptation code has been automatically generated [44], but specialization of other adaptation strategies has not previously been attempted.

3.2 Low-Level Virtual Machine (LLVM)

This work uses the Low Level Virtual Machine (LLVM) [45] infrastructure extensively. LLVM is a compiler framework that includes the ability to create new compiler intrinsics and optimize and generate machine code for several instruction set architectures at runtime. It provides a code representation with type information, explicit control flow graphs, and dataflow. The LLVM framework has a modular structure with extensive support for adding new transformations and analyses as module or function passes. It includes a powerful pass management system used for scheduling passes and, when possible, pipelining them for efficiency. LLVM passes run on LLVM bitcode, which is an instruction-level code representation,

The concurrency compiler presented in this work is constructed as a set of LLVM passes. These passes include analysis passes as well as transform passes. The LLVM code representation allows for powerful analyses which gather information needed to transform the existing code into the EPL representation. That transformation is also done using passes written in LLVM.

Table 3.8: LLVM passes used in this work.

Pass	Description
Promote Memory to Register	Promotes memory references to be register references whenever possible
Scalar Replacement of Aggregates	Breaks up allocation of aggregate types into individual allocations for each member
Combine Redundant Instructions	Combines instructions to form fewer, simple instructions.
Rotate Loops	A simple loop rotation transformation
Sparse Conditional Constant Propagation	Propagates constants
Simplify the CFG (Control Flow Graph)	Performs dead code elimination and basic block merging
Assign Name to Anonymous Instructions	Assigns names to instructions that do not have names
Canonicalize Induction Variables	Transforms each loop to have a single canonical induction variable
Loop Information	Finds all loops in a function and stores useful information about them

Many useful transform and analysis passes are provided by LLVM. This work uses the following passes described in table 3.8: Promote Memory to Register, Scalar Replacement of Aggregates, Combine Redundant Instructions, Rotate Loops, Sparse Conditional Constant Propagation, Simplify the CFG (Control Flow Graph), Assign Name to Anonymous Instructions, Canonicalize Induction Variables, and Loop Information.

Chapter 4

Hierarchy

A major shortcoming of the existing EPL representation is that it did not support hierarchy. Hierarchy exists when a task is only defined entirely within the context of another task, or keeping with the EPL representation, a task descriptor is only defined entirely within the context of another task descriptor. A subtask has no meaning on its own. It must be part of its parent task. In practice, hierarchy is very common. In fact, few programs are written without some sort of hierarchy. Every nested loop introduces hierarchy, as does every function call inside another function.

It is important to note that whether or not hierarchical tasks exist depends on how tasks are defined. For example, if tasks are allowed to contain loops, an inner loop would not need to be defined as a separate, and therefore sub, task. If function calls are allowed inside of a task, a nested function call would not need to be its own task. The term for the different options in defining tasks is task granularity. There is an infinite spectrum of possible task granularity ranging from very coarse to very fine. Coarse grained tasks include many instructions, possibly including function calls and loops. Fine grained tasks include fewer instructions, possibly only a few.

Adjusting task granularity is important in CIRA. In order to adapt most effectively, different granularities are needed for different applications. Much research has already been done in coarse grained parallelism [23, 39, 46] as well as in fine grained parallelism [31, 47, 48]. However, little research has been done with granularities in between fine and coarse. We expect CIRA to demonstrate that many applications work best with some kind of medium granularity. The first step to demonstrate this is to add functionality to the EPL representation for hierarchical task descriptors, thus empowering it to handle finer granularities.

4.1 Hierarchical Task Descriptors

Task descriptors need to contain some extra pieces of information to support hierarchy. First, a subtask descriptor needs to keep track of its parent task descriptor. Often relationships will exist between subtask descriptors of different parents. As stated earlier, however, subtasks are only defined within the context of the parent task descriptor. In order to create a relationship between subtask descriptors of different task descriptors, the context of the parent descriptor is needed to completely define the subtask descriptor. For this reason, knowing which task descriptor is a subtask descriptor's parent is essential. The parent id parameter in the call to TASKDESC provides this information.

It is also important to know which task descriptors are top level descriptors, meaning they have no parent and, therefore, are not subtask descriptors. This information is essential since subtask descriptors need to be handled differently. Also, task descriptors need to keep track of what depth of hierarchy they are at. Top level descriptors are at level 0, their subtask descriptors are at level 1, and so forth. This is necessary for naming task instances which will be described in detail later. The depth parameter in the call to TASKDESC provides this information.

Since parent task descriptors have subtask descriptors, they need to be handled differently than other task descriptors. Though there are additional reasons, the simplest reason we cannot treat all task descriptors in the same way is illustrated by the following example. Consider the pseudocode in figure 4.1. The outer loop becomes the parent task descriptor and the inner loop becomes the subtask descriptor. The parent task descriptor has code before and after the subtask descriptor. The subtask descriptor has a data dependence on the line of code just before it which would become a relationship descriptor. However, the line of code just after the subtask descriptor has a data dependence on the subtask descriptor which would also form a relationship descriptor. If not treated differently, this would result in the parent task descriptor depending on the subtask descriptor as well as the subtask descriptor depending on the parent descriptor which would form a circular dependence.

In order to handle parent task descriptors correctly, we restrict them to have an empty body. Instead the parent task descriptor has sub task descriptors for each part of what would be its body. This results in relationship descriptors between the different subtask descriptors but no circular dependences. This method also keeps the parent task descriptor's instantiation function as simple

```
for(int i=0; i<size1; i++)
{
  int size2 = size1*size1;
  for(int j=0; j<size2; j++)
  {
    arr1[i] += arr2[j];
  }
  arr2[i*size1] = arr1[i];
}
```

Figure 4.1: An example of a piece of hierarchical code that needs to be split up into subtask descriptors.

as possible. Keeping task instantiation functions simple is desirable since it reduces the overhead of task instantiation and also simplifies the adaptation compiler's analysis to choose an adaptation strategy.

As an example of how this works, we once again consider figure 4.1. The outer loop becomes the parent task descriptor. Inside the loop are three sections of code that will each become its own task descriptor. The inner loop is the obvious subtask descriptor. In order to eliminate the code from the body of the parent task descriptor, the rest of the code inside the outer loop needs to be moved to subtask descriptors. The code before the inner loop becomes one subtask descriptor and the code after the inner loop becomes another subtask descriptor. Now the parent task descriptor has no code in its body function and the subtask descriptors contain all the code as shown in figure 4.2. The loop headers are pulled into the instantiation functions.

To keep task descriptors in a standardized form, parent task descriptors still have a body function even though it is empty. Their instantiation functions still have a call to CREATETASK to form task instances of the parent task. This is helpful when forming relationships. The tasks created have the empty body function, so when executed they just return. This method keeps everything in a standardized form which allows for optimizations without introducing too much unnecessary overhead. Having a standardized form also keeps information pertinent to choosing an adaptation strategy in logical places. This should aid the adaptation compiler in its analyses for choosing an adaptation strategy.

```
void original_body(int* arr1, int* arr2, int size1)
{
}

void sub1_body(int* arr1, int* arr2, int size1, int i)
{
  int size2 = size1*size1;
}

void sub2_body(int* arr1, int* arr2, int size1, int i, int j)
{
  arr1[i] += arr2[j];
}

void sub3_body(int* arr1, int* arr2, int size1, int i)
{
  arr2[i*size1] = arr1[i];
}
```

Figure 4.2: Body functions of the original task descriptor and each of the subtask descriptors.

4.2 Hierarchical Task Instantiation

For top level task descriptors, it is simple to instantiate their task instances. The task instantiation function is run. Sub task descriptors are more complicated because their task instances must be instantiated for each task instance of the parent task descriptor. Consider again figure 4.1. For each iteration of the outer loop, an instance of the parent task descriptor is instantiated. Each subtask descriptor needs to run its instantiation function once for each iteration of the outer loop, or in other words, for each task instance of the parent descriptor.

Subtask descriptors need information particular to the parent task instance. The variables holding this information are called parent variables and differ from descriptor time variables and instantiation time variables. The most common parent variables are loop variables such as the *i* variable in figure 4.1 which will have a different value for each instance of the parent task descriptor. Each subtask descriptor will need to have the correct value for *i* depending on which task instance is its parent.

The method we use to call the subtask instantiation functions the correct number of times and to supply the needed parent variables to the instantiated tasks is to call the subtask instantiation

```
void original_inst(int* arr1, int* arr2, int size1)
{
  for(int i=0; i<size1; i++)
  {
    createtask(i, COST, i);
    sub1_inst(arr1, arr2, size1, i);
    sub2_inst(arr1, arr2, size1, i);
    sub3_inst(arr1, arr2, size1, i);
  }
}

void sub1_inst(int* arr1, int* arr2, int size1, int i)
{
  createtask(0, COST, i);
}

void sub2_inst(int* arr1, int* arr2, int size1, int i)
{
  for(int j=0; j<size2; j++)
  {
    createtask(j, COST, i, j);
  }
}

void sub3_inst(int* arr1, int* arr2, int size1, int i)
{
  createtask(0, COST, i);
}
```

Figure 4.3: Instantiation functions of the original task descriptor and each of the subtask descriptors.

functions inside the instantiation function of the parent task descriptor. The parent instantiation function includes the outer loop so that it can call CREATETASK for each instance. The calls to the sub task instantiation functions are placed inside the loop right after the call to CREATETASK, so they are each called once for each parent task instance. All of the parent variables are available and can be passed right into the subtask instantiation functions. The instantiation functions for the above example are shown in figure 4.3. These functions are missing name vector information. This vital information will be added in the next section.

4.3 Task Instance Name Vectors

Task instances need unique identifiers to distinguish them from other instances of the same descriptor. We call these identifiers task instance names. Names for task instances are created inside the instantiation function and passed into the call to CREATETASK. Since there is one call to CREATETASK for each instance, this guarantees that each instance will have a unique name. With the introduction of hierarchy, there is now another use for these names. They can be used to keep track of the context of the task descriptor.

In order to keep track of the descriptor's context, task instance names are stored as vectors of integers. Each element in the vector corresponds to a level of the hierarchy. The number of elements in the vector corresponds to the depth of the task. For example, if a task descriptor has a depth of one, meaning it is one level down from the top, it will have a name vector with two elements. The first element will correspond to the parent task instance and the second element will correspond to its task instance. If it has subtask descriptors, the subtasks' names will be the same as its name except they will have an extra element at the end of the vector for the subtask instance. In figure 4.1 the subtask instance corresponding to iteration 3 of the outer loop and iteration 2 of the inner loop would have the vector name [3,2].

Name vectors are constructed in the task instantiation functions. To construct the name vectors, we create a new name vector with the correct size for the depth. We then load all of the values from the parent name vector and store them into the new name vector. The parent name vector must be passed as a parameter into the subtask instantiation functions. Lastly, the new integer for the subtask instance is stored in the last element.

Vector names have many advantages. As was already described, it is a good way to differentiate between task instances that are subtasks of different parent task instances. In the previous example, the name vectors [2,2] and [3,2] both correspond to the same iteration of the inner loop, but are sub tasks of different parent task instances, and that is made obvious by the name vectors. Name vectors also provide a simple way to keep track of the parent instance, as well as all ancestor instances up to the top level. Even with a deep hierarchy we get task names such as [3,5,2,6,1,3] and it is very easy to see exactly which branch of the hierarchy this task instance is on. Having all of the ancestor instances as part of the name vector is also useful for keeping track of parent variables.

Vector names are also helpful with relationships. When relationship descriptors are instantiated, the full context of each task instance is needed to determine which relationship instances must be created. The full context of a subtask instance includes the context of its parent task instance. Vector names specify which instance of a parent task descriptor is the subtask descriptor's parent. This completes the subtask descriptor's context. For example, when determining if there should be a relationship between task descriptor t1 instance [2,4] and task descriptor t2 instance [3,1], the relationship knows to use the context of instance [2] of t1's parent descriptor and instance [3] of t2's parent descriptor.

Another advantage for relationships is that the construction of the name vector puts in an inherent relationship between parents and subtasks. Although the parent task has no body, and therefore no data dependence or locality relationships, it could still be useful to have that hierarchical relationship to aid the adaptation compiler in its analysis to choose an adaptation strategy. Figure 4.4 shows the instantiation functions for the previous example with the construction of the name vectors added. When the subtask instantiation function loads the name vector of its parent and then stores the values into its own name vector, the load and store instructions are left in an order that will show a data dependence between them. This dependence can be processed similarly to other dependences when determining relationships.

```
void original_inst(int* arr1, int* arr2, int size1)
{
  for(int i=0; i<size1; i++)
  {
    int new_name[1] = {i};
    createtask(new_name, COST);
    sub1_inst(new_name, arr1, arr2, size1, i);
    sub2_inst(new_name, arr1, arr2, size1, i);
    sub3_inst(new_name, arr1, arr2, size1, i);
  }
}

void sub1_inst(int* name, int* arr1, int* arr2, int size1, int i)
{
  int new_name[2] = {name[0], 0};
  createtask(new_name, COST, i);
}

void sub2_inst(int* name, int* arr1, int* arr2, int size1, int i)
{
  for(int j=0; j<size2; j++)
  {
    int new_name[2] = {name[0], j};
    createtask(new_name, COST, i, j);
  }
}

void sub3_inst(int* name, int* arr1, int* arr2, int size1, int i)
{
  int new_name[2] = {name[0], 0};
  createtask(new_name, COST, i);
}
```

Figure 4.4: An example showing how vector names are assigned.

Chapter 5

Concurrency Compiler

The concurrency compiler transforms the application into the EPL representation. It needs to both recognize and use annotation provided by the programmer as well as perform analyses to discover the parallelism in the application. It then needs to construct the task descriptors and relationship descriptors that make up the EPL representation. This includes constructing the task instantiation and body functions and the relationship instantiation functions.

5.1 Building Task Descriptors

For a concurrency compiler to be most useful, it needs to build task descriptors from the existing code with minimal help from the programmer. The programmer should not have to dictate which block of code needs to be turned into a task descriptor or how to instantiate the task instances. Though we do want to allow the programmer the freedom to insert some directives when he chooses to, we should never rely on having directives specifying how to create task descriptors.

We have created an LLVM pass which transforms the code to include task descriptors, task instantiation functions, and task bodies. The only directives that need to be inserted are PARBEGIN, which specifies the beginning of a section to be parallelized, and PAREND, which specifies the end of the section. A programmer could even just insert PARBEGIN at the beginning of the main function and insert PAREND at the end of it, and the whole program would be one parallel section.

5.1.1 Extracting Parallel Sections

The first step to building task descriptors is to extract the parallel sections into their own functions so that we can work with those functions independent of everything else. Since most LLVM passes run on functions, having the parallel sections in their own functions makes it very easy to run LLVM passes on them. This applies to LLVM passes we have written as well as passes

```
EXTRACTPARALLELSECTION()
 1  for each function
 2      Find the call to parbegin
 3      Split the basic block
 4          parbegin is the last instruction in the original basic block
 5          Create a new basic block with all instructions after parbegin
 6      Find the call to parend
 7      Split the basic block
 8          parend is the first instruction in the original basic block
 9          Create a new basic block with all instructions before parend
10      Find all blocks between the parbegin and parend calls
11          This is a simple procedure based on dominator analysis
12      Extract the list of basic blocks into a new function
```

Figure 5.1: Algorithm for parallel section extraction.

that are part of the LLVM source code. This concurrency compiler can only handle parallel sections that only have one entry, the call to PARBEGIN, and one exit, the call to PAREND. Only one parallel section per function is allowed. Future extensions could be to support multiple parallel sections per function as well as branches out of and into parallel sections. The algorithm in figure 5.1 describes the process of extracting parallel sections.

The PARBEGIN and PAREND directives specify the beginning and end of each parallel section. All of the code in between these directives needs to be extracted into its own function. Because it is easier to extract entire basic blocks instead of individual instructions, we first split the basic blocks containing the PARBEGIN and PAREND directives. We split them so that the call to PARBEGIN becomes the last instruction of its basic block and all instructions following it are moved into their own new basic block. Similarly we split the PAREND basic block so that the call to PAREND becomes the first instruction in its basic block and all previous instructions are moved into their own basic block.

Using a simple procedure based on dominator analysis, we determine the set of basic blocks that are between the calls to PARBEGIN and PAREND. We then extract that set of basic blocks into its own function. LLVM provides a function for extracting code sections into functions. We pass it the list of basic blocks and it determines which variables need to be passed into the function. It

```
EXTRACTLOOPSANDBLOCKS()
 1  Run LoopInfo pass
 2  for each loop o
 3      Extract o into its own function f
 4      Split the basic block containing f
 5          All instructions before f in one basic block
 6          All instructions after f in another basic block
 7          f in its own basic block
 8  Order the loop basic blocks using dominator analysis
 9  Find all basic blocks before the first loop basic block
10  Extract the list of basic blocks into a new function
11  Find all basic blocks after the last loop basic block
12  Extract the list of basic blocks into a new function
13  for each pair of consecutive loop basic blocks
14      Find all basic blocks between the two loop basic blocks
15      Extract the list of basic blocks into a new function
```

Figure 5.2: Algorithm for loop and block extraction.

also takes care of any other details for building the function including replacing the extracted code with a call to the function.

5.1.2 Loop and Block Extraction

The EPL representation supports many types of task descriptors, but only two of them are currently supported by our simple adaptation compiler. These two types are loops and blocks. For this reason we have focused on extracting only loops and blocks into task descriptors. In the future we will need to update both this concurrency compiler as well as our adaptation compiler to support all types of task descriptors.

The basic form for extracting loop and block task descriptors is to first extract all loops and then extract the region of code between the loops into blocks. This concurrency compiler requires that the region of code between loops has a single entry, the previous loop, and a single exit, the subsequent loop. Future concurrency compilers could be extended to allow for branches around loops. The algorithm for extracting loops and blocks is shown in figure 5.2. The obvious first step is finding all the loops. We use LLVM's LoopInfo pass to accomplish this. After running

this pass on our parallel section function, we have a list of all the loops in the function. We can then iterate over the list and handle each loop individually. The loop information provided by this pass includes the list of basic blocks contained in the loop, the header block which contains the loop variable, and the block containing the loop variable update and the condition check. It also provides the loop variable as long as there is a canonical induction variable for the loop. We can ensure that there is one by running another LLVM pass beforehand which transforms all loops to have canonical induction variables.

Once we have the list of loops, we can iterate over them and pull each one into its own function. Another LLVM function, much like the code extraction function, accomplishes this. The extracted function includes information for both the task instantiation function and the body function, which are the two functions needed for each task descriptor. Once we have the loops extracted into their own functions, we split the basic blocks containing the calls to those functions so that all loop function calls are in their own basic blocks. Using dominator analysis we can order the loops. Once we have the order of the loops, we can use the same procedure used in parallel section extraction to extract the regions of code before the first loop, after the last loop, and between any two consecutive loops into their own block task descriptors.

Another important piece of information we get from the LoopInfo pass is the number of subloops each loop has. We use that information to determine if we need to turn the loop body into a body function or if we need to turn it into subtask descriptors. When a loop has subloops, we treat it similarly to the parallel section function. We need to find the loops and turn them into loop subtask descriptors and then turn the blocks of code before, after, and between loops into block subtask descriptors. While this is a slightly different process for subtask descriptors because they are inside of a loop, the same basic algorithm is used.

5.1.3 Creating Instantiation and Body Functions

For each extracted loop function which has no subloops we need to extract the body function and leave the extracted function as the instantiation function. If the loop header, including the loop variable creation, the loop variable update, and the loop condition check, was in a separate basic block from the rest of the loop body, it would be easy to extract the body and leave the loop header in the instantiation function. Unfortunately those parts of the loop header often depend

on instructions in the loop body and are all bundled into the same basic block. In fact, it is quite common for variables inside the loop header to be used for other instructions in the body as well as for the loop header. In order to have a fully functional body as well as a fully functional loop header, instructions often have to be copied so they can exist in both functions.

We must perform a careful analysis to determine which instructions inside the loop are needed only in the instantiation function, which are needed only in the body function, and which are needed in both. All instructions that are part of the loop header, including the loop variable creation, loop variable update, and the loop condition, are needed in the instantiation function. Also, any branch instructions that use the loop condition are needed in the instantiation function. All store instructions should be in the body function. One exception is that store instructions that affect the structure of the loop would need to stay in the instantiation function. For example, when the loop variable is stored in memory and updated with store instructions, these store instructions affect the structure of the loop. This concurrency compiler cannot handle these types of store instructions, however, in many cases running the MEM2REG LLVM pass eliminates these store instructions by promoting memory references to be register references. Future concurrency compilers could add support for these types of store instructions.

The algorithm for this transformation is shown in figure 5.3. We start by copying the extracted loop function so that the original can become the instantiation function and the copy can become the body function. For the body function, we need to remove all instructions that are part of the loop header and change branch instructions that branch back to the beginning of the loop to branch to the next basic block after the loop. In this way, the loop is removed. For the instantiation function, we remove all store instructions. We then recursively remove all instructions in both functions that no longer have any uses.

To illustrate this analysis we will consider figure 5.4. The loop is basic block *bb*. The instructions *indvar*, *indvar.next*, and *exitcond* make up the loop header and are deleted from the body function. In the body function the branch instruction is changed to be an unconditional branch to the return basic block. The store instruction is deleted from the instantiation function. Instructions *1* and *3* are only used by the store instruction, so they are also deleted from the instantiation function. Instruction *2* is only used by instruction *3*, so it is also deleted from the instantiation function. The values *A* and *indvar* are also needed by the body function, but we cannot copy the

```
SEPARATEINSTANDBODY()
 1  Copy the extracted loop function
 2      The original becomes the instantiation function
 3      The copy becomes the body function
 4  For the body function
 5      Remove instructions that are part of the loop header
 6      Change the branch instruction to branch out of the loop
 7      Recursively remove instructions that no longer have any uses
 8  For the instantiation function
 9      Remove all store instructions
10      Recursively remove instructions that no longer have any uses
11  Determine descriptor time, parent, and instantiation time variables
12      for each instruction in the body function
13          for each operand to the instruction
14              if it originates from an instruction in the instantiation function
15                  Save it as an instantiation time variable
16              else it must originate from an argument to the instantiation function
17                  if this is a top-level task descriptor
18                      Save it as a descriptor time variable
19                  else this is a subtask descriptor
20                      if the corresponding variable in the parent function is an instruction
21                          Save it as a parent variable
22                      else the corresponding variable in the parent function is an argument
23                          Save it as a descriptor time variable
24  Create the name vector in the instantiation function
25  Create the call to createtask in the instantiation function
```

Figure 5.3: Algorithm for separating the instantiation and body functions.

loop variable to the body function, and *A* is a function argument. In this case we add these two values to the body function's argument list. Figure 5.5 shows the resulting task instantiation and body functions.

After we have extracted the body function from the instantiation function we will be left with some arguments to the body function that originate in the instantiation function. Normally we would just pass in those variables to the call to the body function, but there is no call to the body function. Instead we need to determine which ones are descriptor time variables, which are parent variables, and which are instantiation time variables. If we are working with a top level descriptor, then if the variable in question originates from an argument to the instantiation function, it must be

```
void loop([8 x i32]* %A)
{
entry:
  br label %bb

bb:
  %indvar = phi i32 [ 0, %entry ], [ %indvar.next, %bb ]
  %1 = getelementptr [8 x i32]* %A, i32 0, i32 0
  %2 = load i32* %1
  %3 = add nsw i32 %2, i32 %indvar
  store i32 %3, i32* %1
  %indvar.next = add i32 %indvar, 1
  %exitcond = icmp ne i32 %indvar.next, 3
  br i1 %exitcond, label %bb, label %return

return:
  ret void
}
```

Figure 5.4: An example loop that needs to be separated into a task instantiation function and a task body function.

a descriptor time variable. If it originates from an instruction in the instantiation function, it must be an instantiation time variable. If we are working with a subtask descriptor, however, we need to differentiate between descriptor time variables and parent variables which both originate from the instantiation function arguments. This is accomplished by checking the parent instantiation function and see if the corresponding variable is an argument to that function or an instruction in the function. If it is created in the parent instantiation function, then it is a parent variable. If not, it is a descriptor time variable.

With the information on descriptor time variables, parent variables, and instantiation time variables, we can create the call to CREATETASK inside the instantiation function. The first argument to CREATETASK is the name vector. The name vector is created here in the instantiation function as described earlier. The last arguments to CREATETASK are the parent variables followed by the instantiation time variables. The call to CREATETASK is inserted into the loop in the instantiation function immediately after instructions specifying instantiation variables. In the previous

```
void task_instantiation([8 x i32]* %A)
{
entry:
  br label %bb

bb:
  %indvar = phi i32 [ 0, %entry ], [ %indvar.next, %bb ]
  call createtask(%indvar)
  %indvar.next = add i32 %indvar, 1
  %exitcond = icmp ne i32 %indvar.next, 3
  br i1 %exitcond, label %bb, label %return

return:
  ret void
}

void task_body([8 x i32]* %A, i32 %indvar)
{
entry:
  br label %bb

bb:
  %1 = getelementptr [8 x i32]* %A, i32 0, i32 0
  %2 = load i32* %1
  %3 = add nsw i32 %2, i32 %indvar
  store i32 %3, i32* %1
  br label %return

return:
  ret void
}
```

Figure 5.5: The resulting task instantiation and body functions.

example we omitted the construction of the name vector as well as passing the name vector and the cost into CREATETASK in order to keep the example simple and uncluttered.

In this work we do not handle in and out variables. As stated in chapter 3, they are not necessary for a functional representation. However, they are very useful for increasing performance. In the future, an extra analysis will have to be added to determine in which situations descriptor time or parent variables could be localized and replaced by in and out variables to increase perfor-

mance. We anticipate that the overhead of passing values along relationships will be less than the overhead of accessing a memory location that often will no longer be stored in the cache.

Extracting the body function for block tasks is much easier. There is no loop to worry about, so the whole block of code becomes the body function. The instantiation function just needs to create the name vector and call CREATETASK. There are not any instantiation time variables, so the only other parameters are descriptor time variables and possibly parent variables which are determined in the same way as for loop task descriptors.

5.1.4 Creating Calls to TASKDESC

After extracting loops and blocks, the calls to TASKDESC need to be constructed. Most of the information needed for these calls is gathered while making the instantiation and body functions. The only other information needed is the unique identifier which we can create. The difficult part of this is getting the call for each task descriptor in the right place. They all need to be in order according to the original code order, and TASKDESC calls for parent descriptors must come before TASKDESC calls for subtask descriptors. This is necessary for ordering relationships.

For top level task descriptors we can replace the call to the instantiation function, which was left over when we extracted the loop or block, with the call to TASKDESC. For subtask descriptors, we need to find the call to the top level ancestor task descriptor and insert its call to TASKDESC at the end of that basic block. We use the parent id, which is part of the call to TASKDESC, to determine the top level ancestor task descriptor.

5.2 Determining Relationships

The second piece needed to form a task graph is to determine the relationships between tasks. The relationships allow the scheduler to create a valid ordering of task execution. They are necessary for determining which tasks can be run in parallel and therefore achieve greater performance gains. Relationships are an essential part of the EPL representation.

In the future CIRA will need to handle many different types of relationships. Because data dependence relationships are the most important, and because our adaptation compiler currently only handles data dependence relationships, we have focused primarily on these types of relationships. Our work in automatically generating relationship descriptors only generates data

dependence relationships. This concurrency compiler, as well as our adaptation compiler, will need to be expanded to support other types of relationships in the future.

Once the task descriptors have been formed, we have all the necessary information to create relationship descriptors. Any pair of task descriptors whose task instances could have relationships between them need to have a relationship descriptor relating them. Task instances of the same task descriptor could even have relationships between them. In this case the relationship descriptor would describe what would cause a data dependence between instances of the descriptor. Task instances of different task descriptors could also have relationships between them. In this case the relationship descriptor would describe for each task descriptor what would cause a data dependence with an instance of the other task descriptor. The process of determining which pairs of task descriptors need relationship descriptors will be described in section 5.2.1.

The next step in generating relationship descriptors is creating the relationship instantiation function for each relationship descriptor. This function contains the code that will determine and generate individual relationship instances at runtime. This function needs to be simple enough to avoid adding too much overhead but also complete so that no relationships are ever missed. The process of generating relationship instantiation functions is described in section 5.2.2.

The final step is to create the calls to RELDESC. This is similar to creating the calls to TASKDESC except calls to RELDESC require less information making it a bit more simple. Generating these calls is described in section 5.2.3.

5.2.1 Cross-product of Tasks

Determining which task descriptors could have relationships between them in the EPL representation has significant challenges. The code has been split up into task descriptors and each task descriptor could have multiple task instances. This representation places the code for each task descriptor in its own context. To complicate matters further, each task instance has elements unique to itself. In particular, the instantiation time variables have unique values depending on the task instance. This means that even task instances are in their own contexts. Determining data dependences between two different contexts is impossible without first merging the contexts.

For instructions inside the same task instance, determining data dependences is easier since the instructions are in the same context and all operands to each instruction can be related to one

```
void task_body([8 x i32]* %A, i32 %i)
{
entry:
  %0 = add nsw i32 %i, i32 1
  %1 = getelementptr [8 x i32]* %A, i32 0, i32 %0
  %2 = load i32* %1
  %3 = add nsw i32 %i, i32 2
  %4 = getelementptr [8 x i32]* %A, i32 0, i32 %3
  store i32 %2, i32* %4
  ret void
}
```

Figure 5.6: It is impossible to tell which instances have data dependences between them without knowing the variable *i*.

another. Unfortunately data dependences within a single task instance are useless since the whole task is executed without interruption. All useful data dependences are between instructions of different task instances. As stated earlier, even different task instances of the same task descriptor, which use the same body function, are not really in the same context because the instantiation variables will have different values for each instance. Figure 5.6 demonstrates this problem. The descriptor time variable *A* is an integer array. The instantiation time variable *i* is used for indexing into *A*. The body function has a load and a store to different elements in *A* depending on *i*. Without knowing the value for *i* it is impossible to tell which loads and stores from different instances create data dependences.

This problem gets even more difficult for task instances from different task descriptors. Figure 5.7 demonstrates this case. Task descriptor *t1* has a descriptor time variable *A*, and task descriptor *t2* has a descriptor time variable *B*. Both are integer arrays. Both body functions have memory accesses to the first element in the array, which just increments that element. It would appear that there is no data dependence, except we cannot tell if *A* and *B* point to the same memory location. If they do, then there is a data dependence.

The previous examples show why having instructions in the same context helps immensely in determining data dependences. Because relationship descriptors relate two task descriptors, we attempt to place task instances of two task descriptors in the same context. We do this for every pair of task descriptors. In order to place all instances of each task descriptor in the same context

```
void td1_task_body([8 x i32]* %A)
{
entry:
  %0 = getelementptr [8 x i32]* %A, i32 0, i32 0
  %1 = load i32* %0
  %2 = add nsw i32 %1, i32 1
  store i32 %2, i32* %0
  ret void
}

void td2_task_body([8 x i32]* %B)
{
entry:
  %0 = getelementptr [8 x i32]* %B, i32 0, i32 0
  %1 = load i32* %0
  %2 = add nsw i32 %1, i32 1
  store i32 %2, i32* %0
  ret void
}
```

Figure 5.7: Since we cannot tell if *A* and *B* point to the same memory location, we cannot tell if there is a data dependence here.

we form a kind of cross-product of the pair of task descriptors. We accomplish this by placing the task instantiation function of one task descriptor inside the task instantiation function of the other task descriptor. We call this new function the cross-product function. For instantiation functions that contain loops, each iteration will contain a pair of task instances in the same context. As a simple example to demonstrate how an instantiation function is placed inside another instantiation function to form a cross-product function, consider figure 5.8, which has the two separate instantiation functions, and figure 5.9, which has the cross function. The second instantiation function is placed inside the loop of the first instantiation function so that each task instance created by one instantiation function can be related to each task instance of the other instantiation function.

To merge contexts of the task descriptors in the cross-product function, we need to determine which descriptor time variables are equivalent. For subtask descriptors we also need to determine which parent variables are equivalent. The task instantiation functions contain all of these variables as arguments. The cross-product function has as arguments all of the arguments of

```
void inst1(int* A){
  for (int i = 0; i < 2; i++){
    createtask(i);
  }
}

void inst2(int* B){
  for (int j = 0; j < 2; j++){
    createtask(j);
  }
}
```

Figure 5.8: Two simple instantiation functions which may or may not be equal.

```
void cross(int* A, int* B){
  for (int i = 0; i < 2; i++){
    createtask(i);
    for (int j = 0; j < 2; j++){
      createtask(j);
    }
  }
}
```

Figure 5.9: Cross function for previous instantiation functions.

each task instantiation function, so it, too, has all of the descriptor time and parent variables. Some of these variables may be equivalent.

To determine which descriptor time variables are equivalent we must examine the descriptor time variables passed into each call to TASKDESC in the parallel section function. If the same value was passed to the calls to TASKDESC, then those descriptor time variables must be equivalent. To determine which parent variables are equivalent, we must examine the parent task instantiation functions. Because parent variables are created in the parent task instantiation function, a prerequisite is that the two task descriptors must have a common parent. In cases with multiple levels of hierarchy, the ancestor task where the parent variable was created must be common to both task descriptors. If a common ancestor is found, we must check if the values passed in as parent variables to the calls to the subtask instantiation functions are the same. Once we have determined

```
void cross(int* A, int* B){
  for (int i = 0; i < 2; i++){
    A[i] = A[i+1];
    for (int j = 0; j < 2; j++){
      B[j] = B[j+1];
    }
  }
}
```

Figure 5.10: Cross function with bodies inlined.

which variables in the instantiation functions are equivalent, we can determine if there are possible data dependences between the task descriptors.

Task instantiation functions don't contain the code that could have data dependences. That code is in the body functions. In order to bring that code into the cross-product function, we replace the calls to CREATETASK with the code from the body function, thereby bringing all of the important code from both task descriptors into the cross-product function. Figure 5.10 shows our previous example with the bodies inlined.

Now that all of the necessary code is in the cross-product function we do an intraprocedural backward slice on all pointers used in memory accesses in order to determine any possible data dependences. The way a backward slice works is to take a given value and trace backward through the code to find all other values that this value is dependent on. The values that a given value is dependent on are known as sources for the given value. The backward slice returns a list of sources for each value we run it on. Running a backward slice on the pointers used in memory accesses tells us which memory accesses could be dependent on congruent values – values which refer to the same register or location in memory. This is the information needed by the relationship instantiation function to determine which task instances require relationships between them.

While the list of sources for each memory access pointer would be enough to determine the relationships, it would leave a large amount of work for the relationship instantiation function to do at runtime. It would have to evaluate each of the sources and compare it to each of sources of the other memory accesses. That evaluation could require accessing memory or other things that could drastically reduce performance. This concurrency compiler includes one optimization to reduce that workload. When the backward slice returns descriptor time variables as sources

for two memory access pointers, it can determine if both variables are congruent. It does this by checking the calls to TASKDESC in the parallel section function to see if both values passed in as descriptor time variables are equivalent. While this procedure proves that two memory accesses depend upon the same variable, it does not quite prove that they actually access the same memory location. Many memory accesses also use indices, for example, to specify which element in an array is being accessed. It can be proven that two memory accesses both depend on the same array, but they only need a relationship if they depend on the same element of the array. Since the indices are often based on the instantiation time variables, whose values cannot be known until runtime, the definitive check on which memory accesses are dependent on equivalent values must also wait until runtime. The relationship instantiation functions, described in the next section, contain code to check the indices.

The only index comparisons for determining relationships are in the relationship instantiation function. However, because some indices are not based on instantiation time variables and can be known at compile time, a possible future optimization could be to also examine indices. The goal of this optimization would be to eliminate some relationship descriptors and to simplify the relationship instantiation functions by proving that the indices either cannot be equivalent or must be equivalent. We anticipate that having fewer unnecessary relationship descriptors and more simple relationship instantiation functions will cut down time used in instantiating relationships.

5.2.2 Relationship Instantiation Functions

For each relationship descriptor which we have determined must be created, we need to create a relationship instantiation function. This function will be executed at runtime to instantiate relationship instances. It needs to contain the code which will determine which relationship instances actually exist. As with the cross-product function, the relationship instantiation function will also need the merged context in order to determine which relationship instances should be created. The relationship instantiation function does not need the full task bodies, however. It only needs the values necessary for determining which task instances have data dependences and, therefore, need a relationship instance.

During the optimization described in the previous section, any indices associated with the sources in the trace back to the descriptor time variable were saved along with the sources. These

lists define exactly which indices and sources need to be equivalent for two memory accesses to need a relationship between them.

As an example, consider figure 5.12 which is the LLVM bitcode representation of the cross-product function and figure 5.13 which is the same function in the C programming language. Basic block *bb* contains the code for one task descriptor and basic block *bb2* contains the code for the other. Though there are many memory accesses in this example, we will focus only on the store instructions. After computing the backward slice for each store instruction's pointer operand, we find that each pointer operand has as sources two GETELEMENTPTR instructions, a load instruction, a BITCAST instruction, a loop variable, and a function argument. Assuming the two function arguments, *A* and *B*, are proven to be congruent, these two store instructions could be accessing the same location in memory. All memory accesses in the traces from the store instructions to the function arguments are kept in a list. In this example, the lists would each include the pointer from the store instructions, which is a GETELEMENTPTR instruction, and the pointer from the first load instruction, another GETELEMENTPTR instruction, since it is in the trace back to the function argument. The indices for the sources in the trace are also kept in a list. In this example, the index lists would contain the loop variables and the constant *1*.

The steps required to create the instantiation function are listed in figure 5.11. The first step in constructing the relationship instantiation function is similar to the first step in constructing the cross-product function. We place the task instantiation function for one of the task descriptors inside the task instantiation function of the other task descriptor. However, we do not inline the body functions because the memory alias analysis has already been done.

Next, we use the lists described earlier to build conditional statements which will determine at runtime whether or not to create a relationship instance. The optimization returned which pairs of memory accesses possibly had equivalent pointers. To determine if the pointers are truly equivalent, the list of indices for the first pointer must be equal to the list of indices for the second pointer. If any index from one pointer's list is not equal to the corresponding index from the other pointer's list, the pointers are not equivalent. Conditional statements which will perform this analysis at runtime are inserted into the relationship instantiation function for each pair of possibly equivalent pointers. An additional conditional statement is added to determine if any pair of possi-

CREATERELINST()

```
1  Insert second task instantiation function into the first task instantiation function
2  for each pair of possibly equivalent pointers
3      Create a conditional statement to check if all corresponding indices from the lists are equal
4  Create a conditional statement to check if any of those conditional statements are true
5  Create a branch conditional on this statement to the createrel block
6  for each index used in any of the conditional statements
7      Run the backward slice memory alias analysis on the index
8      Copy over all of the sources that are not already in the relationship instantiation function
9  Create the call to createrel
```

Figure 5.11: Algorithm for creating the relationship instantiation functions.

bly equivalent pointers is indeed equivalent. If even just one of these pairs is truly equivalent, the relationship must exist.

Since the instructions from the body functions are not in the relationship instantiation function, it is possible that some of the indices depend on values that are not in the instantiation function. To resolve this issue we must copy over all needed values from the cross-product function. We once again run the backward slice, but this time on the indices. We copy all of the sources of each index to the relationship instantiation function. This ensures that the conditional statements which were just created are valid. Figure 5.14 shows the relationship instantiation function for the previous example after creating the conditional statement and copying over all needed values. Another optimization which could be considered would be to remove any conditions on constant values. This could result in either a more simple overall condition or completely prove the condition false without needing to check the other index condition statements.

The final step in creating the relationship instantiation function is creating the call to CREATEREL. Most of the parameters are fairly simple because we are only creating data dependence relationships and we are not passing information along the relationships yet. The names are a little more difficult to determine, however, the backward slice helps us once again. The name is passed into the call to CREATETASK in the task instantiation function. We copy that value over to the relationship instantiation function. We run the backward slice on that value for each task descriptor

```
void cross(i8* %A, i8* %B)
{
entry:
  br label %bb

bb:
  %indvar = phi i32 [ 0, %entry ], [ %indvar.next, %bb_exit ]
  %1 = getelementptr i8* %A, i32 1
  %2 = bitcast i8* %1 to i32**
  %3 = load i32** %2
  %4 = getelementptr i32* %3, i32 %indvar
  %5 = load i32 %4
  %6 = add i32 %5, 1
  store i32 %6, i32* %4
  br label %bb2

bb2:
  %indvar2 = phi i32 [ 0, %bb ], [ %indvar2.next, %bb2 ]
  %7 = getelementptr i8* %B, i32 1
  %8 = bitcast i8* %7 to i32**
  %9 = load i32** %8
  %10 = getelementptr i32* %9, i32 %indvar2
  %11 = load i32 %10
  %12 = add i32 %11, 1
  store i32 %12, i32* %10
  %indvar2.next = add i32 %indvar2, 1
  %exitcond2 = icmp ne i32 %indvar2.next, 3
  br i1 %exitcond2, label %bb2, label %bb_exit

bb_exit:
  %indvar.next = add i32 %indvar, 1
  %exitcond = icmp ne i32 %indvar.next, 3
  br i1 %exitcond, label %bb, label %return

return:
  ret void
}
```

Figure 5.12: A cross-product function with multiple memory accesses in its trace to an equivalent function.

```
void cross(int* A, int* B)
{
        for(int i=0; i<3; i++){
                A[i]++;
                for(int j=0; j<3; j++){
                        B[j]++;
                }
        }
}
```

Figure 5.13: The same cross-product function in the C programming language.

and copy all of the sources to the relationship instantiation function. We now have the names as well as all values needed by the names and can pass the names into the call to CREATEREL.

5.2.3 Generating Calls to RELDESC

The calls to RELDESC only have three parameters. The first two are the ids for the task descriptors. Since we know which task descriptors the relationship descriptor is for, we have those ids. The last parameter is the relationship instantiation function which we just created. The final step is placing the calls to RELDESC in the correct place in the parallel section function. The only requirement is that they are placed after the calls to TASKDESC for their task descriptors, so we place them immediately after the later of the two calls to TASKDESC.

```
void relInst(i8* %A, i8* %B)
{
entry:
  br label %bb

bb:
  %indvar = phi i32 [ 0, %entry ], [ %indvar.next, %bb_exit ]
  %4 = getelementptr i32* %3, i32 %indvar
  br label %bb2

bb2:
  %indvar2 = phi i32 [ 0, %bb ], [ %indvar2.next, %bb2_exit ]
  %cond = icmp eq i32 %indvar, i32 %indvar2
  %cond2 = icmp eq i32 1, i32 1
  %cond3 = and %cond, %cond2
  br i1 %cond3, label %yesBB, label %bb2_exit

yesBB:
  // The call to createrel will go here
  br %bb2_exit

bb2_exit:
  %indvar2.next = add i32 %indvar2, 1
  %exitcond2 = icmp ne i32 %indvar2.next, 3
  br i1 %exitcond2, label %bb2, label %bb_exit

bb_exit:
  %indvar.next = add i32 %indvar, 1
  %exitcond = icmp ne i32 %indvar.next, 3
  br i1 %exitcond, label %bb, label %return

return:
  ret void
}
```

Figure 5.14: The relationship instantiation function after creating the condition and moving over needed values.

Chapter 6

Evaluation

The true measure of success for this work is its ability to transform several different applications into the EPL representation. It is not necessary that this first concurrency compiler can successfully transform any type of benchmark into the EPL representation. This work is meant to prove that the parallelism inherent in a serial application can be discovered and exposed automatically with only a few extra directives inserted by the programmer. The most effective construction of this proof is in the form of a working concurrency compiler. This proof is realized with a concurrency compiler that can successfully transform even just a handful of diverse applications. Future concurrency compilers will be charged with the task of expanding this set of applications.

This chapter focuses on the evaluation of the concurrency compiler, which, as previously stated, accomplishes the goals of this work. In addition to successfully transforming applications into the EPL representation, the concurrency compiler can be evaluated with other metrics such as compilation time and generated code size. Future concurrency compilers will necessarily be optimized according to these metrics.

6.1 Methodology

The incomplete nature of the CIRA project introduces difficulties in the evaluation of only a single piece of it. Ideally the evaluation of a concurrency compiler would be done in a complete toolchain. In particular, the adaptation compiler, which is the next step in the toolchain, is still under development. Only one adaptation compiler exists, and it only has one adaptation strategy which is based on the inspector-executor strategy. It doesn't do any sort of analysis to determine the best adaptation strategy. Because of these limitations, the performance of many benchmarks cannot be improved. If the correct adaptation strategy does not yet exist, the adaptation compiler could significantly reduce performance. Despite these limitations, we have been able to complete

an evaluation of the concurrency compiler on several benchmarks of various types. This evaluation includes several metrics which will be described shortly.

6.1.1 Benchmarks

Matrix Multiply The first benchmark is a naive matrix multiply. This benchmark was very useful during development because of its simplicity and directness. It does, however, have one complex element: hierarchy. It contains a triple-nested loop resulting in many levels of hierarchy. The concurrency compiler's ability to transform this benchmark into the EPL representation demonstrates its secure handle on hierarchy.

Jacobi Sparse Iterative Solver When the EPL representation was initially developed in [11] and [10] the benchmark used to evaluate its correctness was the Jacobi sparse iterative solver. The Jacobi solver multiplies a sparse matrix by a dense vector to produce a new dense vector. This new vector is used as the input to the next iteration. The solver continues to iterate until the vector converges. The Jacobi solver was originally chosen because of its simplicity, high potential for parallelism, and presence in many important applications.

For the development of the EPL representation, the transformation from the Jacobi solver application to the EPL representation was done by hand and was not a simple task. The concurrency compiler's ability to automatically generate the EPL representation for the Jacobi solver provides a direct example of automatically discovering and exposing the parallelism in a program to get the same results as if a programmer had done it herself.

Gauss-Seidel Sparse Iterative Solver The Gauss-Seidel sparse iterative solver is very similar to the Jacobi sparse iterative solver. The main difference is in the recurrence equation. More specific differences are described in [11]. The Gauss-Seidel solver has many of the same properties as the Jacobi solver and was also transformed into the EPL representation by hand. Once again this solver provides an opportunity to compare a transformation done by hand and one done by the concurrency compiler.

183.equake, 175.vpr, and jpegdec The first three benchmarks were similar in that they all were used in the development of different pieces of the CIRA project. Even the adaptation compiler was

developed with an adaptation strategy designed for these benchmarks. They are also similar in that they are all quite simple. There was no need for the concurrency compiler to find specific parts of the program that would be most beneficial to parallelize because the whole program was that one main piece. None of these programs are complex enough to merit the parallelization of only specific parts.

An important part of the evaluation of the concurrency compiler is to show that it can parallelize only specific parts of a large and complex program. It is also important that it works on benchmark programs found in well-known and often used benchmark suites. The last three benchmarks used in this evaluation satisfy this requirement.

The first of these benchmarks is 183.equake and is part of the SPEC CFP2000 benchmark suite. It calculates seismic wave propagation. The next benchmark is 175.vpr. It is part of the SPEC CINT2000 benchmark suite and performs FPGA circuit placement and routing. It is important to show that the concurrency compiler can work on both integer and floating point benchmark programs. The last benchmark is jpegdec and is part of the MediaBench benchmark suite. It performs JPEG image compression decoding. All of these benchmark programs were first profiled to determine which sections would be most beneficial to parallelize. Only these sections were transformed into the EPL representation.

6.1.2 Metrics

In addition to the limitations the incomplete CIRA toolchain places on the benchmarks, certain metrics that could be used to evaluate the concurrency compiler are severely limited. These limitations on the metrics are a result of how closely the concurrency and adaptation compilers work together to influence the metrics. For example, the amount of time it takes to instantiate the task descriptor graph is mostly a metric for the adaptation compiler since it defines how, or even if, instantiation will occur at runtime. However, the task descriptor graph is formed by the concurrency compiler. Therefore, a more effective concurrency compiler could lead to better instantiation times. We could also consider the adaptation compiler's job of analyzing the EPL representation in order to choose the best adaptation strategy. With a more effective concurrency compiler, this analysis could be improved and sped up. However, with an adaptation compiler that does not do any analysis to choose an adaptation strategy, this metric becomes impossible to measure. The

following metrics are the most pertinent to the concurrency compiler and the results are the least influenced by the adaptation compiler.

Bitcode File Size This metric evaluates the code size overhead introduced in the generation of the EPL representation. Although code size is not a limiting factor on most systems, it is important that the concurrency compiler does not introduce an inordinate amount of storage overhead. The comparison is done between the bitcode files at different steps of the CIRA flow. A bitcode file is first generated by LLVM. Because the concurrency and adaptation compilers are implemented as series of LLVM passes, they also generate bitcode files. This makes for direct comparison between the original LLVM bitcode, the bitcode after having run the concurrency compiler, and the bitcode after having run the adaptation compiler.

Compilation Time A far more important metric is the time it takes the concurrency compiler to run. This is especially important in research and development environments where an application may need to be changed and recompiled many times during development. There are many important comparisons on compilation time. First is the comparison to the LLVM compilation time. This is the time it takes to compile the original code into LLVM bitcode. We can also compare concurrency compilation time to adaptation compilation time as well as linking time.

6.2 Measurements

The most important measurement is the verification that the application runs successfully after being converted into the EPL representation by the concurrency compiler. All six benchmarks ran successfully and generated the exact same output as the serial versions.

6.2.1 Bitcode Size

To break down how much each piece of the CIRA flow contributes to the increase in code size, the bitcode files have been compared. Figure 6.1 shows the portions of the code size increase that each piece is responsible for. Table 6.1 shows the increase in overall bitcode size caused directly by the concurrency compiler and the increase in overall bitcode size caused directly by the adaptation compiler. This table shows that although the concurrency compiler introduces more

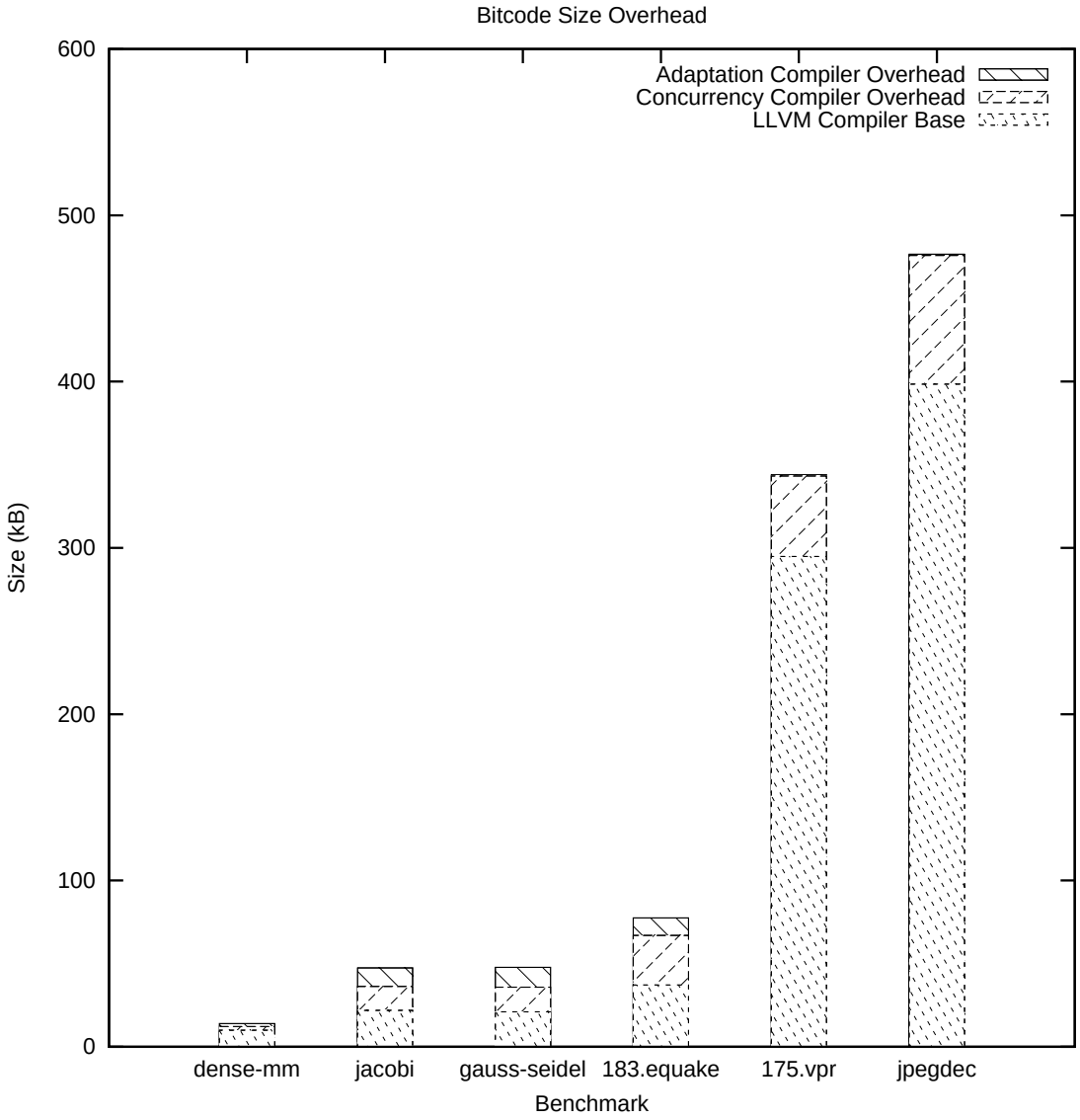

Figure 6.1: Code Size Overhead.

overhead than the adaptation compiler, the overall increase is not very high. In every case the overhead introduced by the concurrency compiler is less than the LLVM base code size, and with larger benchmarks the percentage increase is even lower.

6.2.2 Compilation Time

Perhaps the most important metric for a compiler is the compilation time. To give a good perspective on the overall effect the concurrency compiler has on compilation time we have com-

Table 6.1: Bitcode size increase due to the concurrency compiler and adaptation compiler.

Benchmark	Concurrency Increase	Adaptation Increase
dense-mm	22.941%	14.673%
jacobi	66.619%	30.622%
gauss-seidel	70.524%	33.243%
183.equake	79.690%	15.738%
175.vpr	16.379%	0.201%
jpegdec	19.435%	0.172%

Table 6.2: Increase in compile time due to concurrency compiler, adaptation compiler, and linker.

Benchmark	Concurrency Increase	Adaptation Increase	Linker Increase
dense-mm	18.401%	9.550%	18.622%
jacobi	28.033%	6.859%	11.246%
gauss-seidel	27.791%	7.378%	11.483%
183.equake	126.999%	28.673%	36.916%
175.vpr	69.196%	28.222%	9.033%
jpegdec	67.599%	26.517%	5.558%

pared it to the LLVM compilation time, adaptation compilation time, and linking time. Figure 6.2 gives a good overview of which compilers take the most time. Table 6.2 shows the overall compilation time added by each piece.

For most benchmarks the concurrency compilation time is much shorter than the LLVM compilation time, though longer than the adaptation compile time and the linking time. Because the adaptation compiler does not do any analysis to choose an adaptation strategy, shorter compile times are expected. Although in some cases the concurrency compiler adds a significant amount of compilation time, we expect that with optimizations added to future concurrency compilers to increase efficiency, the compilation time will be reduced sufficiently.

6.2.3 Comparison to Hand-written EPL Representation

The final comparisons used to determine the effectiveness of this concurrency compiler are comparisons to the hand-written EPL representation presented in [11]. The only benchmark with a hand-written EPL representation with reported metrics in [11] is the Jacobi Sparse Iterative Solver. The first comparison is in compilation time shown in table 6.3. As expected, the hand-written EPL

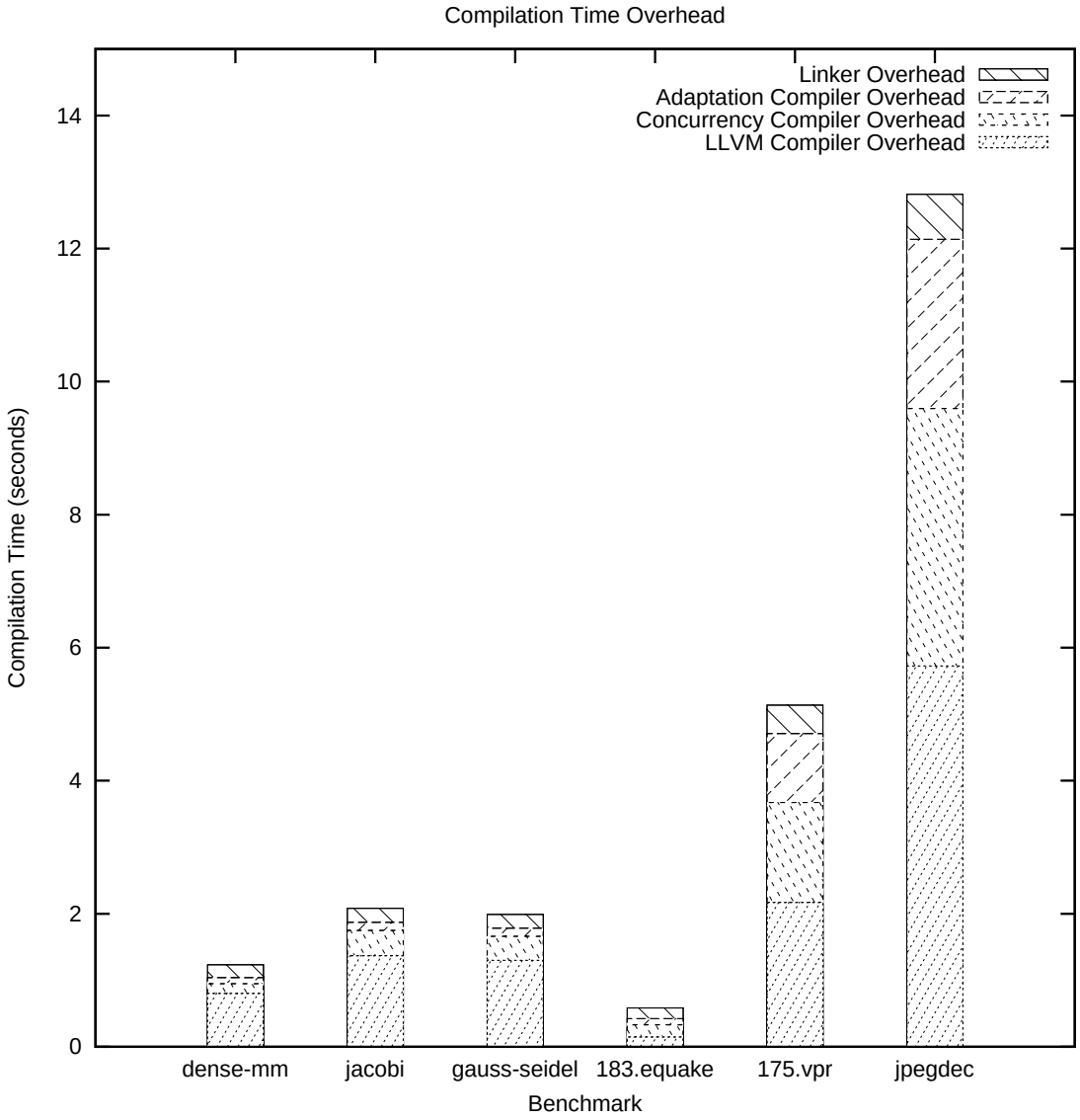

Figure 6.2: Compilation Time Overhead.

representation outperforms the generated EPL representation, and although the difference is more than double, we expect that number to drop as the concurrency compiler is optimized to run more efficiently.

Another interesting comparison is of the actual pieces of the EPL representation. In both cases the task body functions are taken straight out of the application code, so they are not of interest. The task and relationship instantiation functions, however, include more than just the application code. Because of the complexity of the construction of relationship instantiation func-

Table 6.3: Generated compile time compared to hand-written compile time.

Version	Percent Increase Over Serial Version
Hand-Written	3.506%
Generated	11.377%

Table 6.4: Task instantiation function comparison between hand-written and generated versions of the jacobi benchmark.

Metric	Hand-Written	Generated
Average Number of Lines	31.25	30.00
Average Number of Basic Blocks	3.75	2.20
Average Lines per Basic Block	8.33	13.64

tions, a comparison between generated and hand-written relationship instantiation functions can be very telling of the overhead introduced by the concurrency compiler. The comparison is done on the LLVM bitcode. Table 6.4 compares the average number of instructions, number of basic blocks, and instructions per basic block for task instantiation functions of the hand-written and generated version of the jacobi benchmark. Table 6.5 compares the same metrics for relationship instantiation functions of the hand-written and generated version of the jacobi benchmark. The task instantiation functions are actually very similar for all metrics. The generated version appears to have slightly fewer, but larger, basic blocks. As expected the relationship instantiation functions are quite a bit bigger for the generated version. Although the basic blocks are of similar sizes, the generated version has about twice as many.

A final comparison between the hand-written and generated versions of the jacobi benchmark is a comparison of the actual functions. Because of the size difference in relationship instantiation functions, a comparison of these functions could be useful to ensure the generated functions

Table 6.5: Relationship instantiation function comparison between hand-written and generated versions of the jacobi benchmark.

Metric	Hand-Written	Generated
Average Number of Lines	69.33	123.82
Average Number of Basic Blocks	7.00	14.91
Average Lines per Basic Block	9.90	8.30

```
void init0ToSub(inst_it init0, row_struct* row_results1,
  struct jacobi & param1, double tol1, inst_it sub,
  row_struct* row_results2, struct jacobi & param2, double tol2)
{
  for(int i = 0; i < param1.matrix->m; i++)
  {
    int name1[2] = {i,0};
    for(size_t j = param2.matrix->rows[i];
        j < param2.matrix->rows[i+1]; j++){
      int name2[2] = {i,j};
      createrel(name1,0,name2,0,RAW_NODATA, OneToTwo, 0);
    }
  }
}
```

Figure 6.3: Hand-written C code for a relationship instantiation function.

are creating the same relationships and are simple enough to make the adaptation compiler's analysis possible. Figure 6.3 shows the hand-written C code for one of the relationship instantiation functions. Since, only LLVM bitcode is generated, the bitcode must be translated by hand into C-like pseudocode for the comparison. The pseudocode for the equivalent generated function is shown in figure 6.4.

The two functions are very similar, but there are some differences.

- The two versions have different descriptor time variables which leads to different argument lists.
- Despite standard optimizations, the generated function is still not as simple as the hand-written version. For example, the PARENTNAME variables could be optimized out.
- The generated function creates some extra, unnecessary relationships. Showing other relationship instantiation functions would verify that the extra relationships do not affect the ordering of the tasks. An example of the difference is that the hand-written version creates relationships stating that task A needs to precede task B, and task B needs to precede task C. The generated version also creates a relationship stating that task A needs to precede task C, which is already implied by the previous relationships.

```
void init0ToSub(inst_it init0, SparseMatrix* newResult1,
  SparseMatrix* vector1, SparseMatrix* matrix1,
  SparseMatrix* oldResult1,     double tol1, int tmp1,
  int minIterations1, bool* done1,
  inst_it sub, SparseMatrix* newResult2,
  SparseMatrix* vector2, SparseMatrix* matrix2,
  SparseMatrix* oldResult2,     double tol2, int tmp2,
  int minIterations2, bool* done2)
{
  for(int i1 = 0; i1 < matrix1->m; i1++){
    int parentName1[1] = {i1};
    int name1[2] = {parentName1[0], 0};
    for(int i2 = 0; i2 < matrix2->m ; i2++){
      int k=0;
      for(size_t j = matrix2->rows[i2];
          j < matrix2->rows[i2+1]; j++, k++){
        int name2[2] = {parentName2[0], k};
        if(parentName1[0] <= parentName2[0]){
          createrel(name1, 0, name2, 0, RAW_NODATA, OneToTwo, 0);
        }
      }
    }
  }
}
```

Figure 6.4: Pseudo-code for the equivalent generated relationship instantiation function.

6.3 Summary

The concurrency compiler successfully transforms several benchmark programs into the EPL representation. As with all compilers, it adds overhead to both the overall code size and the compilation time. However, the overhead in either case is not excessive and can be reduced with future optimizations. The difference in code size with the hand-written version is also not excessive. In six different benchmark programs, we have shown that a compiler can automatically discover and expose the parallelism in a serial program. Although the metrics on the concurrency compiler gave favorable results, the most important finding in this evaluation is that the goal of this work has been realized.

Chapter 7

Conclusion

This work has shown that parallelism inherent in serial applications can be automatically discovered and exposed. It has demonstrated this proof of concept in the form of the first concurrency compiler for the CIRA project. Additionally, many analyses were developed for this concurrency compiler that will become the basis for many future concurrency compilers. The concurrency compiler has been shown to successfully transform six different benchmark programs into the EPL representation. These benchmarks include a wide variety of application styles. A great deal of hierarchy was present in the benchmark programs, and not only is the concurrency compiler able to recognize the hierarchy, the EPL representation has also been extended to support it.

7.1 Future Work

As with any successful project, the end is simply another beginning. The goals of this work have been accomplished, but there are many other projects that build on this work. There are also numerous ways to expand and improve upon this work. The following is certainly not an all inclusive list, but it does contain the most important and pertinent items.

Support for all types of task descriptors The EPL representation has support for many types of task descriptors which are still not included in the concurrency or adaptation compiler. While loops and blocks cover the basics, the generate EPL representation could be more powerful with the inclusion of other types of task descriptors.

Ability to handle many types of relationships The EPL representation was designed for more than just data dependences. Data dependences are all that's necessary for correct execution, but

for the CIRA project to be successful, it needs to produce significant performance improvement. Additional relationships such as locality relationships could produce that performance increase.

Ability to pass values along relationships Passing values through descriptor time and parent variables is effective, but can become very slow if it results in many cache misses. To reduce that performance penalty the EPL representation has a method to pass values along relationships. This would eliminate many memory accesses that often result in cache misses. We anticipate that extending the concurrency compiler to generate this type of value passing would significantly improve performance.

More robust task descriptor generation Many of the algorithms used to generate the task descriptors had limitations on the code structure they work on. By using better control-flow algorthims making use of dominator trees, post dominator trees, and other analyses, future concurrency compilers could handle many more types of code structures.

Optimize relationship instantiation function The creation of the relationship instantiation function is not currently streamlined. The relationship instantiation function is built to ensure it does not miss any relationships or create an incorrect relationship. Much of the process for creating this function could be optimized. For example, constants can reduce or even eliminate long conditional statements. An optimization based on this concept would simply the relationship instantiation function.

Create better EPL representation Aside from what has already been mentioned, there are always new ideas and new goals to work towards. Many of these could necessitate a more effective construction of the EPL representation. While this work focused on generating a correct EPL representation. Future concurrency compilers will need to ensure they are creating a very effective and efficient EPL representation.

Reduce compilation time and code size overhead Improving the metrics used to evaluate a concurrency compiler will always improve the concurrency compiler. Certainly as concurrency

compilers become more effective they will also require more size and compilation time. Finding ways to reduce those metrics will improve the concurrency compiler.

Increase the number and styles of applications it works on This first concurrency compiler works on six different benchmark programs of varying application styles. However, there are thousands of application styles in the world that concurrency compilers could potentially work on. One of the great strengths of CIRA is that it should be able to work on just about any program. Future concurrency compilers will need to achieve that lofty goal before the entire CIRA project can.

7.2 Summary

The many-core era is on the horizon and programmers are struggling to write parallel programs. The good news is that they should not have to write a completely parallel program. The task for them is being simplified greatly. An understanding aided by a profiler of the parts of the code that would benefit most by being parallelized is all they will need to create parallel programs that will even adapt to the environment and resources available. This is the future that CIRA could create. One of the very first milestones has been achieved. This work has shown that parallelism inherent in serial applications can be automatically discovered and exposed by a compiler.

Bibliography

[1] D. A. Penry, "Multicore diversity: A software developer's nightmare," *Operating Systems Review*, vol. 43, no. 2, pp. 100–101, Apr 2009.

[2] K. Kennedy, "Compiler technology for machine-independent parallel programming," *International Journal of Parallel Programming*, vol. 22, no. 1, pp. 79–98, Feb 1994.

[3] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[4] S. J. Fink, S. B. Baden, and S. R. Kohn, "Efficient run-time support for irregular block-structured applications," *Journal of Parallel Distributed Computing*, vol. 50, no. 1-2, pp. 61–82, 1998.

[5] J. H. Merlin, S. B. Baden, S. J. Fink, and B. M. Chapman, "Multiple data parallelism with HPF and KeLP," in *HPCN Europe 1998: Proceedings of the International Conference and Exhibition on High-Performance Computing and Networking*. London, UK: Springer-Verlag, 1998, pp. 828–839.

[6] F. Bodin, P. Beckman, D. Gannon, S. Narayana, and S. X. Yang, "Distributed pC++: Basic ideas for an object parallel language," *Scientific Programming*, vol. 2, no. 3, pp. 11–20, Fall 1993.

[7] F. Bodin, P. Beckman, D. Gannon, S. Yang, S. Kesavan, A. Malony, and B. Mohr, "Implementing a parallel C++ runtime system for scalable parallel systems," in *Proceedings of the Supercomputing '93*, 1993, pp. 588–597.

[8] L. V. Kale and S. Krishnan, "CHARM++: A portable concurrent object oriented system based on C++," in *Proceedings of the 20th Annual ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages and Applications*, 1993, pp. 91–108.

[9] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy, "Introduction to the Cell microprocessor," *IBM Journal of Research and Development*, vol. 49, no. 4/5, pp. 589–602, July/September 2005.

[10] D. A. Penry, D. J. Richins, T. S. Harris, D. Greenland, and K. D. Rehme, "Exposing parallelism and locality in a runtime parallel optimization framework," in *Proceedings of the 7th ACM International Conference on Computing Frontiers*, 2010, pp. 117–118.

[11] K. D. Rehme, "An internal representation for adaptive online parallelization," Master's thesis, Brigham Young University, Provo, UT, August 2009.

[12] S. Amarasinghe, J. M. Anderson, M. S. Lam, and C. Tseng, "An overview of the SUIF compiler for scalable parallel machines," in *Proceedings of the 7th SIAM Conference on Parallel Processing for Scientific Computing*, 1995.

[13] D. J. Kuck, R. H. Kuhn, D. A. padua, B. Leasure, and M. Wolfe, "Dependence graphs and compiler optimizations," in *Proceedings of the 8th International Symposium on Architectural Support for Programming Languages and Operating Systems*, 1981, pp. 207–218.

[14] F. Allen, M. Burke, R. Cytron, J. Ferrante, W. Hsieh, and V. Sarkar, "A framework for determining useful parallelism," in *Proceedings of the 2nd International Conference on Supercomputing*, 1988, pp. 207–215.

[15] D. Callahan and K. Kennedy, "Analysis of interprocedural side effects in a parallel programming environment," *Journal of Parallel and Distributed Computing*, vol. 5, no. 5, pp. 517–550, 1988.

[16] V. Balasundaram and K. Kennedy, "A technique for summarizing data access and its use in parallelism enhancing transformations," in *Proceedings of the 1987 Symposium on Interpreters and Interpretive Techniques*, 1989, pp. 41–53.

[17] E. Su, D. J. Palermo, and P. Banerjee, "Processor tagged descriptors: A data structure for compiling for distributed-memory multicomputers," in *Proceedings of the 3rd International Conference on Parallel Architectures and Compilation Techniques*, 1994, pp. 123–132.

[18] A. Lain, "Compiler and run-time support for irregular computations," Ph.D. dissertation, University of Illinois at Urbana-Champaign, Champaign, IL, USA, 1996.

[19] D. R. Chakrabarti, P. Banerjee, and A. Lain, "Evaluation of compiler and runtime library approaches for supporting parallel regular applications," in *Proceedings of the 12th International Parallel Processing Symposium*, 1998, p. 74.

[20] K. A. Faigin, S. A. Weatherford, J. P. Hoeflinger, D. A. Padua, and P. M. Petersen, "The Polaris internal representation," *International Journal of Parallel Programing*, vol. 22, no. 5, pp. 553–586, 1994.

[21] *OpenMP Application Program Interface*. OpenMP Architecture Review Board, 2005.

[22] R. P. Wilson, R. S. French, C. S. Wilson, S. P. Amarasinghe, J. M. Anderson, S. W. K. Tjiang, S. Liao, C. Tseng, M. W. Hall, M. S. Lam, and J. L. Hennessy, "SUIF: an infrastructure for research on parallelizing and optimizing compilers," *SIGPLAN Not.*, vol. 29, no. 12, pp. 31–37, 1994.

[23] M. W. Hall, J. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S. Liao, E. Bugnion, and M. S. Lam, "Maximizing multiprocessor performance with the SUIF compiler," *Computer*, vol. 29, no. 12, pp. 84–89, 1996.

[24] J. Zhao, I. Rogers, C. Kirkham, and I. Watson, "Loop parallelization for the Jikes RVM," in *Proceedings of the Sixth International Conference on Parallel and Distributed Computing, Applications, and Technologies*, 2005, pp. 35–39.

[25] E. Yardımcı and M. Franz, "Dynamic parallelization and mapping of binary executables on hierarchical platforms," in *Proceedings of the 2006 ACM International Conference on Computing Frontiers*, 2006, pp. 127–137.

[26] J. H. Saltz, R. Mirchandaney, and K. Crowley, "Run-time parallelization and scheduling of loops," *IEEE Transactions on Computers*, vol. 40, no. 5, pp. 603–612, May 1991.

[27] L. Rauchwerger, "Run-time parallelization: Its time has come," *Parallel Computing*, vol. 24, no. 3–4, pp. 527–556, July 1998.

[28] T. Johnson, "A concurrent dynamic task graph," in *Proceedings of the 1993 International Conference on Parallel Processing*. Washington, DC, USA: IEEE Computer Society, 1993, pp. 223–230.

[29] C. D. Polychronopoulos, "The hierarchical task graph and its use in auto-scheduling," in *Proceedings of the 5th International Conference on Supercomputing*, 1991, pp. 252–263.

[30] H. J. Sips and K. van Reeuwijk, "An integrated annotation and compilation framework for task and data parallel programming in Java," in *Proceedings of ParCo*, 2003, pp. 111–118.

[31] A. Ghuloum, T. Smith, G. Wu, X. Zhou, J. Fang, P. Guo, B. So, M. Rajagopalan, Y. Chen, and B. Chen, "Future-proof data parallel algorithms and software on Intel Multi-Core Architecture," *Intel Technology Journal*, vol. 11, no. 4, pp. 333–347, November 15 2007.

[32] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," *Journal of the ACM*, vol. 46, no. 5, pp. 720–748, September 1999.

[33] M. Wolfe, "More iteration space tiling," in *Proceedings of the 1989 ACM/IEEE Conference on Supercomputing*, 1989, pp. 655–664.

[34] M. K. Chen and K. Olukotun, "Exploiting method-level parallelism in single-threaded java programs," in *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques*, 1998, pp. 176–184.

[35] M. Franklin and G. S. Sohi, "ARB: A hardware mechanism for dynamic reordering of memory references," *IEEE Transactions on Computers*, vol. 45, no. 5, pp. 552–571, May 1996.

[36] L. Rauchwerger and D. A. Padua, "The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelism," in *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, 1995, pp. 218–232.

[37] M. K. Chen and K. Olukotun, "The Jrpm system for dynamically parallelizing Java programs," in *Proceedings of the 30th International Symposium on Computer Architecture*, 2003, pp. 434–446.

[38] M. Herlihy and J. E. B. Moss, "Transactional memory: architectural support for lock-free data structures," in *Proceedings of the 20th International Symposium on Computer Architecture*, 1993, pp. 289–300.

[39] M. A. Suleman, M. K. Qureshi, and Y. N. Patt, "Feedback-driven threading: Power-efficient and high-performance execution of multi-threaded workloads on CMPs," in *Proceedings of the 13th International Symposium on Architectural Support for Programming Languages and Operating Systems*, 2008, pp. 277–286.

[40] M. Kandemir, T. Yemliha, S. Muralidhara, S. Srikantaiah, M. J. Irwin, and Y. Zhang, "Cache topology aware computation mapping for multicores," in *Proceedings of the ACM SIGPLAN 2010 Conference on Programming Language Design and Implementation*, 2010, pp. 74–85.

[41] Y. Ding, M. Kandemir, P. Raghavan, and M. J. Irwin, "A helper thread based EDP reduction scheme for adapting application execution in CMPs," in *Proceedings of the International Parallel and Distributed Processing Symposium 2008*, 2008, pp. 1–5.

[42] R. D. Blumofe, C. F. Joerg, B. C. Kuzmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system," *Journal of Parallel and Distributed Computing*, vol. 37, no. 1, pp. 55–69, August 1996.

[43] C. D. Polychronopoulos, "Toward auto-scheduling compilers," *Journal of Supercomputing*, vol. 2, pp. 297–330, 1988.

[44] L. Rauchwerger, N. M. Amato, and D. A. Padua, "A scalable method for run-time loop parallelization," *International Journal of Parallel Programming*, vol. 26, no. 6, pp. 537–576, July 1995.

[45] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis and transformation," in *Proceedings of the International Symposium on Code Generation and Optimization*, 2004, pp. 75–86.

[46] J. E. Miller, H. Kasture, G. Kurian, C. Gruenwald III, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal, "Graphite: A distributed parallel simulator for multicores," in *Proceedings of the 16th International Symposium on High-Performance Computer Architecture*, 2010, pp. 295–306.

[47] D. K. Lowenthal, V. W. Freeh, and G. R. Andrews, "Using fine-grain threads and run-time decision making in parallel computing," *Journal of Parallel and Distributed Computing*, vol. 37, pp. 42–54, 1996.

[48] S. Kumar, C. J. Hughes, and A. Nguyen, "Carbon: Architectural support for fine-grained parallelism on chip multiprocessors," in *Proceedings of the 34th International Symposium on Computer Architecture*, 2007, pp. 162–173.