Compilation for Explicitly Managed Memory Hierarchies

Timothy J. Knight   Ji Young Park   Manman Ren   Mike Houston
Mattan Erez   Kayvon Fatahalian
Alex Aiken   William J. Dally   Pat Hanrahan
Stanford University

Abstract

We present a compiler for machines with an explicitly managed memory hierarchy and suggest that a primary role of any compiler for such architectures is to manipulate and schedule a hierarchy of bulk operations at varying scales of the application and of the machine. We evaluate the performance of our compiler using several benchmarks running on a Cell processor.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors–Compilers, Optimization; C.1.4 [*Processor Architectures*]: Parallel Architectures–Distributed architectures

General Terms Performance, Design, Experimentation

Keywords Software-managed memory hierarchy, bulk operations

1. Introduction

The advances in semiconductor technology that have dramatically increased the performance possible on a single chip have also undermined the classical random-access model of memory: the idea that a processor can access every memory address in a mostly uniform, and tolerable, amount of time. Instead, there is a large and still growing gap between the processing capacity of functional units and the available global on-chip and off-chip memory bandwidth needed to supply those functional units with data. Additionally, the latency to access off-chip memory and large on-chip memory structures is growing when compared with arithmetic throughput. For decades the standard solution to this problem has been to bridge the gap with hardware-managed caches.

An emerging class of high performance architectures, including the Sony/Toshiba/IBM Cell Broadband Engine Processor$^{\text{TM}}$ [32] (Cell), the ClearSpeed CSX600 [11], and academic projects such as Stanford's Imagine and Merrimac [12, 24], seek to achieve much higher performance and efficiency by exposing a hierarchy of distinct memories managed explicitly in software. Machines with an explicitly managed memory hierarchy are distinguished from conventional cache architectures by three key characteristics. First, processing is highly parallel: multiple high-peak-performance *processing elements* (PEs) execute in isolation entirely out of a local level of the memory hierarchy. Second, individual PE local memories are not virtualized by hardware address translation—no PE has direct access to all of memory and there are no data caches. Programs executing out of these local memories have a fixed, and relatively small, space in which to work and it is the program's responsibility to manage the physical layout and movement of data and, on some architectures, even the layout and movement of the program's code. Third, there are multiple levels to the memory hierarchy, typically at least a conventional high-latency and low bandwidth external memory at the root and multiple simplified PEs with small, fast local storage at the leaves. The movement of data and code between levels of the memory hierarchy is performed exclusively via asynchronous block transfers explicitly orchestrated by software. We believe that machines with such explicitly managed memory hierarchies will become increasingly prevalent in the future.

This paper presents the design and implementation of an optimizing compiler for architectures with software-managed memory hierarchies. We model programs as *hierarchies of bulk operations* with explicit parallelism. Bulk operations are either bulk data transfers or *kernels* (bulk computations) on bulk data. Our focus here is on the backend compiler phases; we begin with a relatively low-level intermediate representation close to the target class of machines. In this representation all parallelism is explicit, every bulk operation is assigned to a specific level of the memory hierarchy, and all sizes of data sets are known. We then illustrate important optimizations that should be applied to such programs to produce efficient code for hierarchical memory machines. In this paper we do not consider the problem of mapping a source-level language to the intermediate representation (but see [16] for an example). Any performance-oriented compiler must address the issues we consider regardless of programming model; a compiler for a programming model further removed from the target class of machines simply faces a superset of the issues we address.

We make the following contributions:

- We present a general model for machines with explicitly managed memory hierarchies (Section 2). This model illustrates the level of abstraction at which our compiler works and provides the conceptual framework in which we develop our optimizations.
- We present an *intermediate representation* (IR) expressing programs as hierarchies of bulk operations. An IR that represents operations at multiple scales enables optimizations to be applied uniformly across different levels of the machine: the same optimizations apply to coarse-grained computation near the root and finer-grained computation at the leaves. We also give an operational semantics for IR programs executing on an abstract machine, which both makes clear our notions of hierarchical programs and bulk operations and provides the basis for understanding the correctness of our optimizations (Section 3).

This work is supported in part by the Department of Energy under the ASCI Alliances program (Contract LLNLB523583), in part by National Science Foundation Graduate Research Fellowships, in part by an Intel Foundation Graduate Fellowship, and by donations from the IBM Corporation.

- We describe a number of optimizations that are key to achieving high performance on our target class of machines (Section 4).
- We discuss some important details of compiling for the Cell architecture (Section 5) and give experimental results with a Cell-based compiler showing the effectiveness of our optimizations for utilizing execution and bandwidth resources (Section 6).

2. Abstract Machine Model

We represent a machine as *tree of memories* T. A given level of the memory hierarchy has a memory M and zero or more sub-machines $T_1, \ldots, T_n$:

$$\begin{array}{rcl} T & := & \langle M, C\rangle \\ C & := & [T_1, \ldots, T_n] \quad n \geq 0 \end{array}$$

A level with no sub-machines $\langle M, []\rangle$ is a *leaf* of the hierarchy. For simplicity, we model a memory M as a function from *names* to values. (We could also use the conventional mapping from addresses to values, but with the exception of data layout most of our optimizations do not depend on actual address values or the size of memory.) Names can be scalar variable names or aggregate structures such as arrays. Values that are definitely scalar are written lower-case ($i, j, k, \ldots$), and values that may be either scalar or aggregate are written upper-case ($A, B, C, \ldots$); we sometimes go beyond simple names and refer to array ranges $A[r]$ in examples. Note that different memories at different levels of the hierarchy have distinct name spaces; the only way that data can be shared between memories is via explicit copies between adjacent hierarchy levels.

Each level of a machine hierarchy has storage and may also have the ability to perform computation. Computation can take place in parallel among siblings in the tree, and, for some machines, there is even some parallelism within a single node. In Section 3 we formalize where and when computation can take place in the form of an operational semantics for programs.

Finally, we chose to model machines as trees of memories as this is a simple, lowest-common-denominator model that can capture important features of a wide range of machines, in particular the performance-critical inter-memory-level communication that we focus on here. While a full discussion is beyond the scope of this paper, we note in passing that we have previously successfully used the tree of memories abstraction to model machines as diverse as a cluster of workstations and Cell blades by introducing additional levels into the tree to represent communication links between elements in the same hierarchy level [16].

3. Programs

In this section we first give the syntax of the low-level IR programs that our optimizing compiler takes as input; to make the model of computation concrete we also give an operational semantics for these programs. Programs for execution on the abstract hierarchical machine are also hierarchical:

$$\begin{array}{rcl} G & := & \texttt{Copy}_{M_i,M_j}(\langle A_1, \ldots, A_n\rangle, \langle B_1, \ldots, B_n\rangle) \quad n \geq 1 \\ & | & \texttt{Kernel}_M(A = f(B_1, \ldots, B_n)) \quad n \geq 0 \\ & | & \texttt{Scalar}_M(a = f(b_1, \ldots, b_n)) \quad n \geq 0 \\ & | & \texttt{If}_M(pred, G_1, G_2) \\ & | & \texttt{For}_M(k = start : end, G) \\ & | & \texttt{Forall}_M(k = start : end, G) \\ & | & \texttt{Group}_M(H) \\ & | & \texttt{Exec}^I_M(G) \end{array}$$

Many of these operations are familiar from conventional programming languages. Note, however, that every form is subscripted with one or more memories M which show the level(s) of the memory hierarchy where that operation executes. For example, an $\texttt{If}_M$ is executed by the processor associated with memory M. Only `Copy` (which moves data between two adjacent levels of the memory hierarchy) and `Exec` (which spawns a new computation at a child of the current level) refer to more than one memory hierarchy level. Note the absence of a sequential composition operator. Programs are dependence graphs giving a partial order on execution; the form $\texttt{Group}_M(H)$ assigns a dependence graph H to a level M of the machine. We assume the reader is familiar with dependence graphs and for brevity we do not formally define them; many examples of dependence graphs are given in Section 4. To define the semantics of programs we use three operations: predicate $Empty : H \rightarrow Bool$ is true if its graph argument is empty, $Head : H \rightarrow 2^G$ returns an arbitrary set of roots of H (i.e., statements ready to execute) and $Rest : H \times 2^G \rightarrow H$ removes a set of roots from the graph, returning the remaining graph.

To define what IR programs mean, we give an operational semantics with rewrite rules of the form

$$T, M \vdash G \rightarrow T', M'$$

which is read: If the initial state of a computation is a hierarchical memory T with parent memory M, then if the program G terminates normally, the result is a modified hierarchical memory T' with modified parent memory M'. Intuitively, G is a computation taking place at level T of the machine; the parent memory M of T is needed for operations in G moving data to or from M. For uniformity we assume the root of the memory hierarchy also has a parent memory; the parent of the root holds the inputs to and records the outputs from the entire program.

Representative operational rules are given in Table 1. We briefly discuss each rule. A `Copy` copies A_j in memory M_i to B_j in memory M_{i+1}. Copy operations can copy multiple data objects in one statement; because copies between different parts of a machine are often expensive it is usually better to copy multiple objects at the same time rather than use multiple copies moving one object each. We adopt the common convention that smaller indices refer to memories closer to the leaves of the hierarchy; thus, the given statement copies data from a child M_i to the parent M_{i+1}. Reversing the roles of M_{i+1} and M_i copies data from parent to child. Also possible are intra-memory copies within memory M_i. We use the standard notation $M[B \leftarrow A]$ for a memory M modified at point B to return the value A.

A `Kernel` executes a bulk computation at memory M_i, with arguments being either scalars or array blocks also in M_i. Our compiler is not concerned with the details of the `Kernel`'s computation—it is just some function of the arguments B_j producing A, although kernels are assumed to be coarse-grained computations (e.g., containing a loop or loop nest or other time-consuming computation). We rely on a standard optimizing compiler to generate good code for a `Kernel`. The semantics of a `Scalar` (not shown) is similar to a kernel operation but only operates on scalars. An `If` is a conventional `if-then-else`, evaluating one of the two branches based on the value of its scalar predicate; the rule for the `else` branch is shown and the rule for the `then` branch is similar. An `If` is an example of a rule with hypotheses; the interpretation is that if the executions above the line hold then the execution below the line also holds. Note all parts of the `If` take place in the same memory.

A `Group` executes a dependence graph in a particular memory. The rewrite rule for `Group` selects a set of statements X of graph G with no predecessors (i.e., all dependences are satisfied) to execute. The rule in Table 1 allows asynchronous execution of statements in X; this feature is necessary to model coarse-grain parallelism within a single processing unit (for example, many architectures provide the ability to overlap a DMA request with

$$\langle M_i, C\rangle, M_{i+1} \vdash \texttt{Copy}_{M_i,M_{i+1}}(\langle\ldots,A_j,\ldots\rangle,\langle\ldots,B_j,\ldots\rangle) \rightarrow \langle M_i, C\rangle, M_{i+1}[\ldots,B_j \leftarrow M_i(A_j),\ldots] \qquad \texttt{[Copy]}$$

$$\langle M_i, C\rangle, M_{i+1} \vdash \texttt{Kernel}_{M_i}(A = f(\ldots,B_j,\ldots)) \rightarrow \langle M_i[A \leftarrow f(\ldots,M_i(B_j),\ldots)], C\rangle, M_{i+1} \qquad \texttt{[Kernel]}$$

$$\frac{\begin{array}{c} M_i(\mathit{pred}) = \mathit{false} \\ \langle M_i, C\rangle, M_{i+1} \vdash G_2 \rightarrow \langle M_i', C'\rangle, M_{i+1}' \end{array}}{\langle M_i, C\rangle, M_{i+1} \vdash \texttt{If}_{M_i}(\mathit{pred}, G_1, G_2) \rightarrow \langle M_i', C'\rangle, M_{i+1}'} \qquad \texttt{[If]}$$

$$\frac{\begin{array}{c} \mathit{Empty}(H) = \mathit{false} \\ \mathit{Head}(H) = X, \quad \mathit{Rest}(H,X) = H' \\ \forall G_j \in X.\ \langle M_i, C\rangle, M_{i+1} \vdash G_j \rightarrow \langle M_i^j, C^j\rangle,\ M_{i+1}^j \\ \langle M_i', C'\rangle = \sum_j^{\langle M_i, C\rangle} \langle M_i^j, C^j\rangle \qquad M_{i+1}' = \sum_j^{M_{i+1}} M_{i+1}^j \\ \langle M_i', C'\rangle, M_{i+1}' \vdash \texttt{Group}_{M_i}(H') \rightarrow \langle M_i'', C''\rangle,\ M_{i+1}'' \end{array}}{\langle M_i, C\rangle, M_{i+1} \vdash \texttt{Group}_{M_i}(H) \rightarrow \langle M_i'', C''\rangle,\ M_{i+1}''} \qquad \texttt{[Group]}$$

$$\frac{\forall k : \mathit{start} \leq k \leq \mathit{end}.\ \langle M_i[j \leftarrow k], C\rangle, M_{i+1} \vdash G \rightarrow \langle M_i^k, C^k\rangle, M_{i+1}^k}{\langle M_i, C\rangle, M_{i+1} \vdash \texttt{Forall}_{M_i}(j = \mathit{start} : \mathit{end}, G) \rightarrow \sum_k^{\langle M_i, C\rangle} \langle M_i^k, C^k\rangle,\quad \sum_k^{M_{i+1}} M_{i+1}^k} \qquad \texttt{[Forall]}$$

$$\frac{\begin{array}{c} \forall k \in I.\ \langle M_{i-1}^k[\mathit{myid} \leftarrow k], C^k\rangle, M_i \vdash G \rightarrow \langle M_{i-1}^{k'}, C^{k'}\rangle, M_i^k \\ \forall k \notin I.\ \langle M_{i-1}^{k'}, C^{k'}\rangle = \langle M_{i-1}^k, C^k\rangle \text{ and } M_i^k = M_i \end{array}}{\begin{array}{l} T = \langle M_i, [\langle M_{i-1}^1, C^1\rangle, \ldots, \langle M_{i-1}^n, C^n\rangle]\rangle \\ T, M_{i+1} \vdash \texttt{Exec}_{M_i}^I(G) \rightarrow \langle \sum_{1\leq k\leq n}^{M_i} M_i^k, [\langle M_{i-1}^{1'}, C^{1'}\rangle, \ldots, \langle M_{i-1}^{n'}, C^{n'}\rangle]\rangle, M_{i+1} \end{array}} \qquad \texttt{[Exec]}$$

Table 1. IR execution rules

computation). The sub-executions, when executed in isolation in the same initial state $\langle M_i, C\rangle, M_{i+1}$, produce different final states $\langle M_i^j, C^j\rangle, M_{i+1}^j$. We must take care to define what it means when multiple computations execute simultaneously. If no computation updates a value another reads, and both write different parts of memory, then the final state is well-defined. The assumption that neither updates a value the other reads is expressed by the separate execution in the same initial state. To reconcile the final states, we say that memories M' and M'' can be *merged with respect to* M, written $M' +_M M''$, if M' and M'' modify disjoint portions of M:

$$(M' +_M M'')(A) \quad = \quad \begin{array}{ll} M'(A) & \text{if } M(A) = M''(A) \\ M''(A) & \text{if } M(A) = M'(A) \\ \text{undefined} & \text{otherwise} \end{array}$$

The extension to hierarchical memories is straightforward. Let $T = \langle M, [\ldots, T_i, \ldots]\rangle$ and $T' = \langle M', [\ldots, T_i', \ldots]\rangle$ and $T'' = \langle M'', [\ldots, T_i'', \ldots]\rangle$. Then

$$T' +_T T'' = \langle M' +_M M'', [\ldots, T_i' +_{T_i} T_i'', \ldots]\rangle$$

Finally, we extend binary sums to n-ary sums in the natural way:

$$\sum_{1\leq i\leq n}^{M} M_i = M_1 +_M \ldots +_M M_n$$

An n-ary sum of memory hierarchies is defined similarly.

A `For` (resp. `Forall`) is a sequential (resp. parallel) loop. Table 1 gives the semantics of `Forall`, as it is the more interesting case. The intention of a `Forall` is that all iterations of the loop can be executed in any order or in parallel. Thus we define the result as the merge of all the final states produced by running each individual iteration separately in the initial state at the same level of the memory hierarchy. This semantics expresses the assumption that all iterations of a `Forall` are independent; our compiler uses this assumption but does not check it.

An `Exec` is the mechanism by which a processor can spawn new computations at lower levels of the memory hierarchy: the operation executes in memory M_i, and its IR subgraph executes SPMD-style in its children at level T_{i-1}. Each `Exec` starts a new SPMD-style execution on the subset of the children given by the indices in I. As is standard in SPMD execution, the *myid* variable in each child is distinct, allowing operations to be selectively executed on only some of the processors. The use of I to restrict the set of children on which an operation is launched allows programs that start two or more distinct SPMD computations on different sets of child nodes. As with other rules above, in the rule for `Exec`, the merge of stores in the conclusion simply says that the computation is defined only if child computations write disjoint portions of the parent's memory.

We conclude this section with a discussion of what it means for an optimization to be correct in our model.

DEFINITION 3.1 (Correctness). Consider programs G and G'. We say G' is a *correct transformation* of G if for every output A of G whenever

$$T, M \vdash G' \to T', M'$$

then

$$T, M \vdash G \to T'', M''$$

and $M'(A) = M''(A)$. Thus, a correct transformation preserves the outputs in the parent memory but not necessarily any other state of the memory hierarchy. Also, a correct transformation may insert dependences into a program's IR, yielding a program which will have fewer possible executions than the original program (i.e., be more deterministic).

4. Machine Independent Transformations

This section presents the machine independent transformations that our compiler performs. Many of these are recognizable as standard compiler optimizations recast for hierarchical memory, but unlike traditional optimizations that may affect a few individual program statements, a single transformation manipulating one or two bulk operations can result in a radical change in the amount of computation performed.

4.1 Copy Elimination

The first three optimizations are referred to collectively as *copy elimination*: each attempts to transform the IR to remove an unnecessary copy operation. Copies may either arise from inefficiencies introduced in translating from source to an IR program or be exposed by other optimizations.

Intra-memory copy elimination, illustrated in Figure 1, eliminates a copy within a single memory module. In presenting our optimizations we depict programs as dataflow graphs, which allows us to visually highlight the flow of data and control, the difference between data and operations, and the level of the memory hierarchy where each operation/datum resides. We stress, however, that this graphical syntax is equivalent to the text syntax used in Section 3, and that an equivalent (but less concise) operational semantics can be given for the graphical representation. In Figure 1, a subrange of *A* is copied to *B* in the same memory module, and a subrange of *B* is subsequently used as an input for operation *OpX*. The compiler performs the following steps: (1) it removes both the copy operation and the data object *B* from the IR, and (2) it adds a scalar operation to compute the range of *A* that corresponds to the data that is input to *OpX*. This transformation saves both the time to execute the copy and the space used to store it. Note that for correctness, there must be no other uses of *B* in the program.

The second copy optimization is *spill copy elimination*, depicted in Figure 2. The object *C* contains a subrange of the original object *A* that was "spilled" from memory M_i to its parent memory M_{i+1} and then copied back. Spill copy elimination transforms the two copies up and down the memory hierarchy into a single intra-level copy, which may itself be eliminated by a subsequent application of the intra-memory copy elimination optimization. This optimization greedily prioritizes the exploitation of producer-consumer locality over the reduction of data object live ranges.

The third type of copy elimination, illustrated in Figure 3, is *duplicate copy elimination*. This optimization detects when an object is copied twice from a parent to a child memory and eliminates the second copy, transforming the IR so that the operations using the data share the copy.

4.2 Operation Hoisting

Operation hoisting, also known as *loop-invariant code motion*, moves operation *Op* from inside to outside a loop if *Op* does not depend either directly or indirectly on the loop's iteration variable. In the example IR in Figure 4, *OpX* takes as input a range of the data object *B* which is a function of the iteration variable, *k*, and thus it cannot be hoisted outside the loop. *OpY*, on the other hand, reads the same input *A* and performs the same computation on every loop

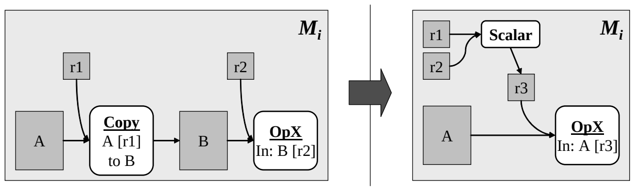

Figure 1. Intra-memory copy elimination

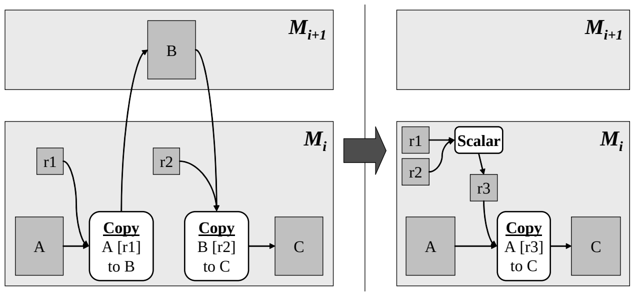

Figure 2. Spill copy elimination

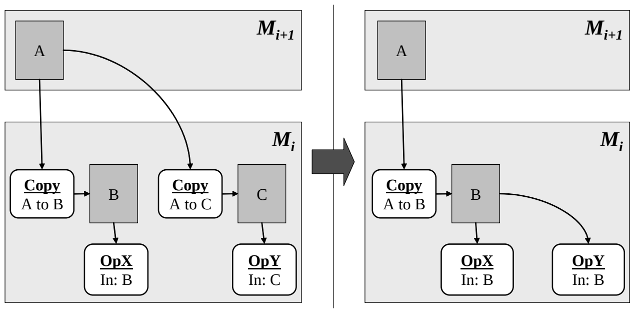

Figure 3. Duplicate copy elimination

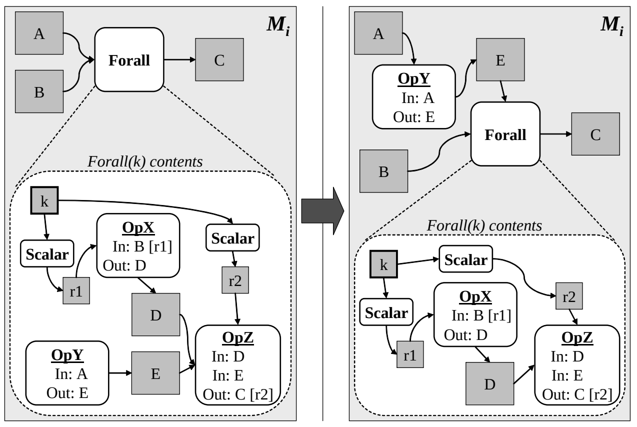

Figure 4. Operation hoisting

iteration, and hence it can be hoisted outside the loop, resulting in a transformed program IR in which *OpY* executes just once.

4.3 SPMD Distribution

If an Exec contains a Forall the compiler can spread the iterations over more or fewer child nodes of the hierarchy depending on what resources are available. Figure 5 presents an example of the IR transformation that occurs when the compiler SPMD-izes a Forall for a single node over eight processors at the same level of the memory hierarchy. First, the compiler introduces a new *myid* variable uniquely identifying each child memory to the program. Second, the Forall is given a new loop variable k' which iterates over one-eighth of the original range. Third, a scalar operation is inserted into the IR to compute the value of the original loop variable as a function of the new loop variable; in this example, the expression is $k \leftarrow myid \cdot t/8 + k'$.

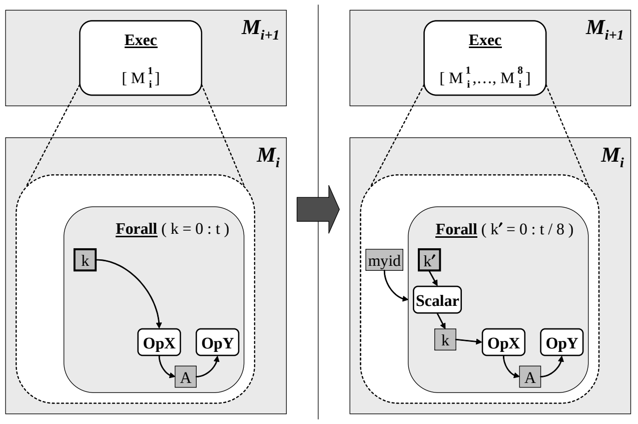

Figure 5. 8-way SPMD distribution of a Forall operation

A single Exec operation expresses the SPMD execution of a set of operations over a set of child processors, but note that neither the IR nor the compiler are restricted to only executing operations in a SPMD fashion. Rather, multiple distinct Exec operations may execute concurrently, each over a different set of child processors, yielding an MPMD execution of a program.

4.4 Operation Scheduling

The semantics given in Section 3 allows an implementation to select any execution order satisfying IR dependences. The *scheduling* transformation, depicted in Figure 6, chooses an operation ordering to optimize some metric, such as execution time. Our current implementation uses a simple set of scheduling heuristics:

- asynchronous operations are scheduled as early as possible,
- operations dependent on asynchronous operations are scheduled as late as possible, and
- similar operations are placed together to increase opportunities for subsequent grouping optimizations (see below).

Once the compiler has determined an operation ordering, it inserts dependences to enforce this ordering. In Figure 6, the compiler has decided that *OpX* should execute before *OpY*. Because the IR is a hierarchical graph, the compiler must schedule the operations within each IR subgraph.

One property not expressed in our IR is any machine resource constraints that affect asynchronous execution. For example, in Figure 6, if *OpX* can execute asynchronously and there are sufficient resources, then it should be overlapped with the execution of *OpY* since its result (*B*) isn't needed until after *OpY* completes, when *OpZ* begins. Our scheduling optimization also takes a *machine description* specifying for each IR operation whether it can be issued asynchronously on the target machine and how many asynchronous operations may be simultaneously in flight. The compiler uses this information in determining operation ordering.

4.5 Copy Grouping

A property of the machines our compiler targets is that a single large copy, which translates into a DMA command or other hardware mechanism, is much more efficient than several small copies. While the compiler's IR is designed to manipulate bulk operations, there are always some number of small copy operations (e.g., moving scalar control variables from one memory to another). The *copy grouping* optimization aggregates independent copy operations into a single copy. Figure 7 illustrates the case where the copies are from a parent M_{i+1} to a child M_i; the resultant bulk copy corresponds to a *gather* operation. Copies from child to parent correspond to a *scatter* operation.

4.6 Exec Grouping

An Exec operation invokes the execution of an IR subgraph on one or more child processors. The best implementation of an Exec depends on the details of the target machine, but typically the means for a processor to start a computation on another processor is relatively expensive. The *exec grouping* optimization, illustrated in Figure 8, merges independent Exec operations into a single Exec, resulting in a larger amount of work to do within a single Exec and lowering the relative overhead. The IR subgraphs in the transformed Exec are grouped and executed in their original order; the compiler inserts dependences to ensure this.

One implementation subtlety arises from the possibility that the Exec may be parallel over a number of child processors. The semantics of the original IR is that the first Exec runs to completion on all child processors before any child processor begins executing the second Exec; there is an implicit barrier between the Execs. The compiler inserts barriers between the Groups in the transformed IR during code generation (not illustrated here).

A second issue arises if two Exec operations with differing SPMD ranges are merged; that is, if one Exec is SPMD-ized over a different set of child processors than the other Exec. The compiler handles this case by giving the transformed Exec the *union* of the SPMD ranges of the two individual Exec operations, and then enclosing each subgraph inside the transformed Exec by an If that is guarded by the *myid* variable (see Section 4.3), ensuring that each IR subgraph is only executed by the same child processors as in the original IR.

Finally, a third issue is that the target machine may only have a small, fixed space to stage the program's code binary. Because the IR subgraph contained in a single Exec effectively corresponds to a block of code that will be issued as a single computational unit, the compiler should not always greedily group Exec operations together. There is a tradeoff: "small" Exec operations have a higher relative overhead due to the cost of making thread calls between

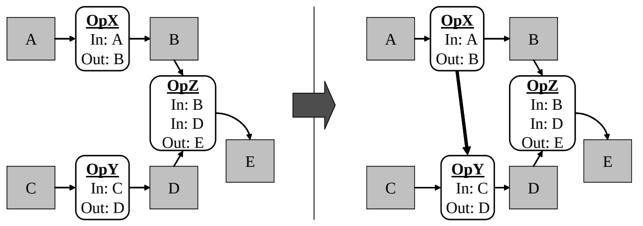

Figure 6. Operation scheduling

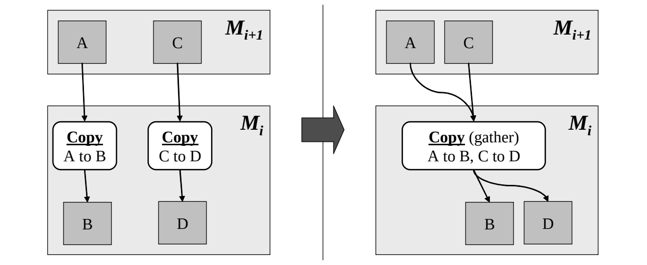

Figure 7. Copy grouping

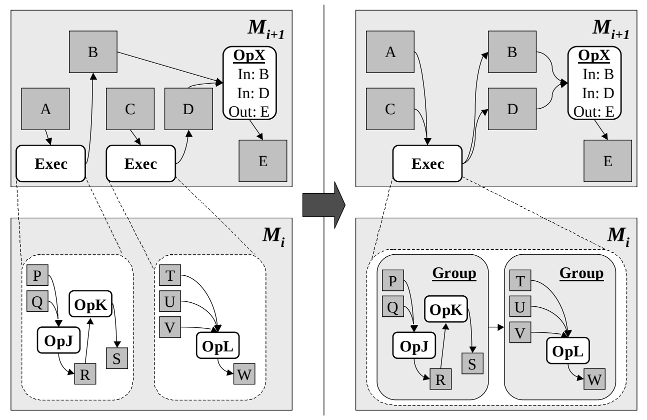

Figure 8. Exec grouping

processors, while "large" `Exec` operations consume more code space, potentially reducing the space available for the program's data. The compiler uses a simple heuristic to navigate this tradeoff: it assumes `Kernel` operations are heavyweight computational units and does not group two `Exec`s together if they each contain a `Kernel`; otherwise, it greedily groups `Exec` operations to the extent possible.

5. Implementation

5.1 Cell Hardware Background

In this paper we use the Cell processor as a case study for an architecture with an explicitly managed memory hierarchy. The Cell has 8 processing elements (called SPEs in IBM's terminology), each with 4 single-precision SIMD floating point and integer arithmetic units, a functional unit that executes branches and manipulates values in the 128-entry, 128-bit wide local register file (permute, load and store instructions), a 256KB local store (holding both data and instructions), and a *Memory Flow Controller* (MFC) DMA unit for bulk requests to memory staged through the local store. The MFC performs 16-byte aligned DMA-style accesses to sequential blocks that are a multiple of 16 bytes in length. Many such requests can be combined into a single asynchronous bulk operation via the *DMA list* mechanism ([22]–Subsection 7.4). DMA lists are traversed by the MFC, which asynchronously issues a request for each specified memory address in order, and synchronizes with the arithmetic units in the SPE once the entire list is processed. The MFCs access an XDR-DRAM based memory system. In addition to the 8 SPEs the Cell also contains a PowerPC processing core that can make arbitrary accesses to off-chip memory through a hardware managed cache hierarchy. Cell's Element Interconnect Bus can be used for horizontal SPE to SPE communication though in this paper we only utilize this feature for inter-SPE synchronization since none of our benchmarks exhibit a speedup from such communication. The exploitation of inter-PE communication paths within a machine modeled as a tree of memories is a subject of ongoing research.

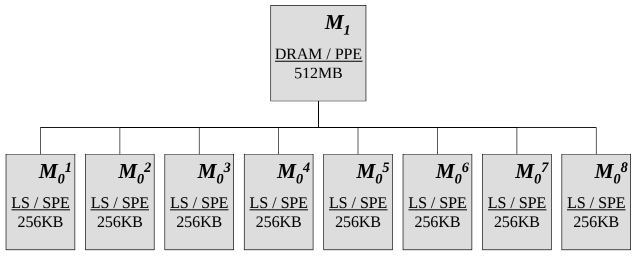

Figure 9. Cell processor's abstract machine model

Figure 9 depicts the abstract machine model corresponding to the Cell processor used in our compiler evaluation. It is a tree in which the off-chip DRAM is the root and the 8 LSes are its children.

5.2 Compiler Implementation

In this section we describe the implementation of a compiler based on the IR execution model presented in Section 3 and which features the IR-to-IR transformations detailed in Section 4. Figure 10 shows our compiler's various passes and their internal ordering with respect to each other. The three copy elimination and the operation hoisting optimizations are applied iteratively; the compiler will continue to apply them until it has applied all four in succession without any of them resulting in the program's IR being transformed, at which point it will move on to the SPMD distribution pass.

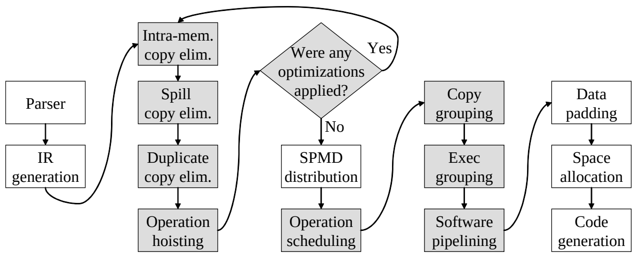

Figure 10. Phase ordering of compiler passes; white boxes are passes that are required for correct operation, constituting the minimum compilation infrastructure needed to just get a program to run, and grey boxes are optimizations which aim to boost program performance

5.2.1 Compiler Front-end

Our compiler accepts programs written in the *Sequoia* [16] programming language. Sequoia is a language which was designed to enable portable, high-performance programming of machines with explicitly managed memory hierarchies. It makes explicit the hierarchy of bulk computations and bulk data transfers that our compiler's IR is centered around. Syntactically, Sequoia is essentially C with some language extensions. Note that due to constraints imposed by the Sequoia programming language, in particular the fact that programs are expressed as a hierarchy of isolated units of execution, our compiler does not need to worry about pointer aliasing, memory consistence, or any of the other factors that make traditional C code difficult for compilers to target to parallel, distributed machines such as Cell.

We modified the ELSA [29] C/C++ parser to accept Sequoia programs and produce a standard AST data structure. This AST is then converted to our compiler's IR in a two-step process: (1) each task (function) in the program is individually converted to its own IR comprising input and output data objects and a (hierarchical) graph of operations and local data objects, and then (2) the compiler statically coalesces the individual task IRs through task callsites to produce specialized IR graphs which span multiple functions in the task call graph. The IR-to-IR compiler transformations described in Section 4 are applied within a single IR graph.

5.2.2 Cell Machine Description Specification

The scheduling optimization (Section 4.4) utilizes a machine description file that each compiler target must provide. The Cell machine description file expresses that `Copy` operations executed by the SPE between the Cell's LS and DRAM may be issued asynchronously with other operations, and in particular with `Kernel` operations, as the Cell SPE hardware provides an asynchronous DMA controller (the *MFC* unit).

5.2.3 Data Padding

Correct execution on Cell requires all arrays to be padded to a multiple of 16 bytes in length in their innermost (row) dimension, ensuring that each row begins on a 16-byte boundary. Further, all data transfers (copies) must be padded to a 128-byte granularity for increased DRAM throughput. Cell's memory system is designed to achieve peak performance when transfers are large and aligned, with short and/or non-aligned transfers achieving significantly lower DRAM bandwidth.

The compiler implements data padding as an IR-to-IR transformation which comprises the following steps: (1) every data object in the IR (containing an annotation detailing its size) is padded by changing these annotations to reflect the above constraints, and (2) any `Scalar` expression that computes the range of a data object that an operation uses will be updated such that the operation will use the same logical data range, despite the data object having been physically increased in size. Data objects may be dynamically sized, in which case their size-related IR annotations refer to program variables rather than constants.

5.2.4 Software-Pipelining

Our compiler implements a simple software-pipelining transformation taking a `Forall` operation on the LS/SPE level of the machine and running multiple iterations in parallel, overlapping operations from different iterations for increased hardware utilization. The software-pipelining algorithm comprises the following steps:

- the dependence graph, now linearized by scheduling (Section 4.4), is split into stages such that any operation that waits on the completion of an asynchronous operation X is in a later pipeline stage than X;
- for a resulting N-stage pipeline, all variables with scope internal to the `Forall` are replicated N times so that N loop iterations can be run in parallel (*modulo variable expansion* [26]);
- the `Forall` loop is transformed by enclosing its contained IR nodes in `Group` operations, one per pipeline stage; and
- the `Forall` loop is emitted in the following manner:

```
for ($k'$ = $start$ : $end - N + 1$) {
    Group$^1$[$k \leftarrow k' + N - 1$];
    Group$^2$[$k \leftarrow k' + N - 2$];
    ...
    Group$^N$[$k \leftarrow k' + 0$];
}
```

The compiler also emits the appropriate partial iterations to prime and drain the pipelined loop (not illustrated).

On Cell, the aim of software-pipelining is to overlap the asynchronous DMA transfers that implement the $\texttt{Copy}_{M_1,M_0}$ and $\texttt{Copy}_{M_0,M_1}$ operations with $\texttt{Kernel}_{M_0}$ operations, and given that there are only two resources (the SPE processor, that executes the kernels, and the MFC unit, that issues DMAs), the maximum possible performance improvement due to software-pipelining on Cell is 2x. Further, in the event that one of either kernel computation or memory transfers dominates the runtime of the application, the gain from software-pipelining will be significantly less than 2x.

Note that as with all other optimizations in this paper, software-pipelining operates at the granularity of bulk operations, overlapping coarse-grained data transfers with large computational kernels. As such, it is distinct from any software-pipelining loop transformations that may be applied by low-level compilers such as GCC, since these operate at the granularity of program instructions.

5.2.5 Space Allocation

The allocation phase reserves the space where each data object resides for its lifetime in an explicitly managed memory module. Unlike cache-based architectures, logically contiguous data must be stored physically contiguously in the local memory, and so the problem of allocating space for data objects is bin-packing a two-dimensional space-time grid. One grid axis represents the contiguous range of addresses consumed by the data object and the other axis represents the interval of time it is allocated there. An example is depicted in Figure 11, in which the IR snippet's packing diagram (on the right) is interpreted as follows: an operation, such as *OpX*, must have all of the data that it is using (its inputs, outputs, and local data) simultaneously resident in the memory throughout its execution. *OpX* requires *C* and *D* to be resident, in this example. If an operation refers to data objects in multiple memory levels, then these data objects must similarly be resident in the appropriate memories throughout the operation's execution.

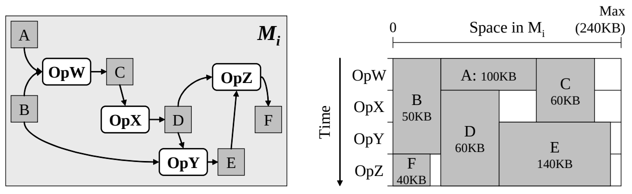

Figure 11. Space allocation

Our packing heuristic is adapted from [28] and is described below. We first compute the live ranges of data objects using their scheduled order, generate the *interference graph* (nodes are data objects and edges imply overlapping live ranges), and initially allocate all objects to address 0 for the duration of their live range. We then greedily modify the allocation such that no two objects share the same space at the same time. We select the next object to *finalize* using the following priorities: the object has the lowest address; overlaps with the largest number of other objects; requires the most space. The object is finalized, shifting the allocation of all objects it interferes with to higher addresses. This process is repeated until all objects are finalized. Packing may fail if an object's allocation exceeds the space of the memory module, at which point the compiler provides feedback to the programmer, who must then make the appropriate modifications to either their source program or the compiler's command-line options.

For Cell, in addition to allocating and packing data objects, allocation also ensures data objects are 16-byte-aligned and reserves space in the local stores for the compiled SPE code.

5.2.6 Code Generation

The final phase of our source-to-source compiler, code generation, comprises the emission of the C code that will be compiled by the target's low-level compiler. The compiler emits two sets of C output files for Cell, one to be compiled using GCC 4.0.2 for Cell's PowerPC control processor (PPE) core and the other to be compiled by IBM's XLC compiler for the SPEs.

`Exec` operations executing on the control processor make thread calls to execute operations on the SPEs. The cost of launching a new Cell SPE thread for each SPE operation using the libspe SDK's `spe_thread_create` proved prohibitively costly, and so to avoid using system threads we implemented event loops on the SPE's. The compiler creates a single system thread on each SPE at program start, which idles until it receives a command from the control processor to execute a block of SPE code. When such a call occurs, the SPE executes the specified code, synchronizes (if necessary) with the other SPEs using Cell's inter-SPE communication, and returns to idling when done, waiting for the next control command. These control commands are issued using Cell's *mailbox* mechanism ([22]–Subsection 8.6).

This approach allows operations to be launched onto the SPEs with modest overhead. However, compiling all operations into a single SPE binary often results in a code text region consuming a large fraction of, or even exceeding, each SPE's 256KB local store. To minimize code footprint while still allowing general code to be executed, we compile the IR subgraph within each SPE `Exec` into a separate SPE overlay. Overlays are loaded dynamically by an SPE from off-chip memory when it receives a message from the control processor (PPE) to execute a block of code. Local store space is reserved for the overlay code text in the space allocation phase of the compiler (recall Section 5.2.5).

Overlay loading is not overlapped with SPE computation since experiments showed that the tiny gain in SPE utilization from such a strategy (for our benchmarks) was offset by a very large performance loss due to the reduction of local store (LS) space available for program data. Typically, Sequoia programs compiled for Cell yield a small number of large overlays, rather than numerous small overlays, due to the optimized kernel code taking a large amount of space.

6. Evaluation

Our evaluation focuses on the utility of the compiler and the effectiveness of its transformations. We implemented seven benchmarks in Sequoia (described in Table 2), compiled them with different optimization combinations, and collected execution statistics on an IBM BladeCenter fitted with a blade containing a Cell processor operating at 2.4GHz accessing 512MB of system memory (see Section 5.1). Subsections 6.1 and 6.2 report the overall performance results for the benchmarks and Subsection 6.3 evaluates the effects of compiler optimization.

6.1 Raw Performance Measurements

Table 3 presents the raw performance numbers measured for each of our seven benchmarks on a single Cell processor. [1] Putting these raw numbers into perspective via direct comparisons with other Cell compilers is difficult, as there are very few Cell compilers available. IBM's Octopiler [15, 14] and the Barcelona Supercomputing Center's CellSs [3] compiler are both optimizing compilers that automatically exploit application parallelism to target the Cell processor, however raw performance numbers have yet to be published from either project, making a comparison impossible. RapidMind Inc. [33] is developing a commercial programming system for Cell in which the programmer expresses parallel operations over collections of data, with a runtime system aggregating these operations into bulk computations and distributing them over the Cell's SPEs, but we are not aware of any published Cell performance results using RapidMind's tools for any of our benchmarks.

[1] These raw performance results were also listed in [16], though small differences are present due to changes in the OS, kernel, firmware, compiler, and library versions installed on our Cell blades. Further, a small amount of additional hand-tuning of kernels was performed here.

SAXPY	BLAS L1 saxpy performed on 32 million word vectors.
SGEMV	BLAS L2 sgemv using a 8192x4096 matrix.
FFT3D	Discrete Fourier transform of a complex 256^3 dataset. Complex data is stored in struct-of-arrays format.
SGEMM	BLAS L3 sgemm, multiplying matrices of size 4096x4096.
CONV2D	Convolution of a 5x5 filter with a 4096x8192 input signal.
GRAVITY	An $O(N^2)$ N-body stellar dynamics simulation on 8192 particles for 100 time steps. We are using Verlet update and the force calculation is acceleration without jerk [19].
HMMER	Fuzzy protein string matching using Hidden Markov Model evaluation. The Sequoia implementation of this algorithm is derived from the formulation of HMMER-search for graphics processors given in [21] and is run on a large fraction of the NCBI non-redundant database.

Table 2. Benchmarks used for evaluation

SAXPY	2.8GFLOP/s	**CONV2D**	57.8GFLOP/s
SGEMV	9.1GFLOP/s	**GRAVITY**	83.3GFLOP/s
FFT3D	45.3GFLOP/s	**HMMER**	9.9GFLOP/s
SGEMM	96.3GFLOP/s		

Table 3. Measured raw performance of each benchmark (single precision)

Given the lack of comparison data from other Cell optimizing compilers, we use the best known hand-tuned implementations of our benchmarks as a basis for comparison. IBM's large FFT implementation runs at 46.8GFLOP/s on a 3.2GHz Cell processor [9], a result comparable to the 45.3GFLOP/s that our implementation of **FFT3D** achieves on a 2.4GHz Cell. Mercury Computer Systems published a 116GFLOP/s performance result [10] for a 1D FFT of a 64K-element dataset on a 3.2GHz Cell, however this dataset size was chosen to perfectly fit the Cell's on-chip local stores and thus avoids the cost of cycling data through off-chip DRAM; our FFT implementation operates on a 16M-element dataset, as does IBM's.

Computing at a rate of 3.5 billion interactions per second, our **GRAVITY** implementation on Cell is 52% faster than the custom hardware of GRAPE-6A [19]. Finally, our Cell implementation of **HMMER** is 5% faster than ClawHMMER [21], an implementation of the same program on an ATI 1900XT graphics processor. ClawHMMER was demonstrated to outperform hand-tuned CPU implementations.

6.2 Hardware Utilization Measurements

Our compiler is designed to optimize a hierarchy of bulk operations and attempts to maximize the utilization of hardware resources. The compiler's ability to schedule the operations is apparent in the results presented in Figure 12, in which **SGEMM**, **CONV2D**, **GRAVITY**, and **HMMER** are compute limited and fully utilize the SPEs' arithmetic resources by running the program's computational kernels for close to 100% of the execution time. The only way to improve the performance of these benchmarks would be to further fine-tune the kernels, which is outside the scope of our compiler; we assume that a `Kernel` is an externally-provided unit that will be executed as-is. For **SGEMM**, for example, we use IBM's `liblarge_matrix` routine as the kernel.

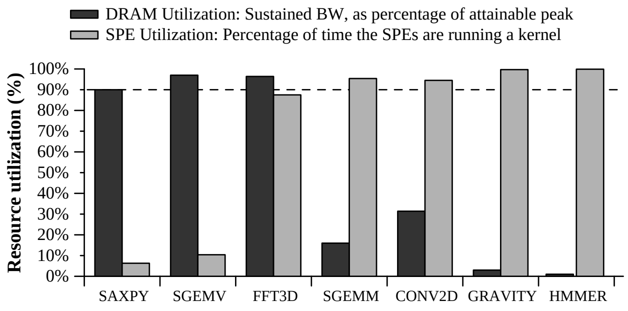

Figure 12. Measured utilization of Cell's compute and bandwidth resources for each benchmark

In contrast, **SAXPY**, **SGEMV** and **FFT3D** are bandwidth limited and our compiler schedules operations in a manner that is within 10% of the optimal DRAM throughput of our Cell system (17.5GB/s). As there are no duplicate or otherwise unnecessary memory transfers being performed in any of these programs (which we verified by inspecting the compiler's output), the resource utilization graph demonstrates that the compiler is executing the bandwidth-limited benchmarks at or near the highest performance possible on the target machine.

6.3 Evaluation of Optimizations

Table 4 details the number of times each optimization was applied on each benchmark, and Figure 13 illustrates the loss in performance when optimizations are progressively disabled: each benchmark's left-most bar corresponds to its measured runtime with all optimizations enabled, and the bars to its right depict the resulting execution times with optimizations successively disabled, normalized to the all-optimizations-on runtime. The scale on the graph is non-uniform to emphasize differences in the 1x to 2x range while providing a dynamic range of 1000x+.

We now discuss the results in Figure 13 in more detail. The copy grouping optimization makes little difference for Cell programs because the hardware DMA controller performs copy grouping by issuing the program's copy operations as simultaneously-in-flight DMAs. `Exec` grouping was clearly effective, however, increasing the performance (measured in GFLOP/s) of four of the benchmarks by 13–175%.

Software-pipelining increased three benchmarks' performance by 30–47%. The software-pipelining implementation is built on top of the scheduling transformation, which orders operations to maximize concurrency. With software-pipelining disabled, the difference between a naive schedule and an efficient schedule was at most 4% in any benchmark's performance.

Intra-memory copy elimination aided **FFT3D** and **GRAVITY**, both of which exhibited intra-main-memory copies resulting from the "copy in, copy out" calling semantics of the source language. The results clearly demonstrate the necessity of successfully optimizing away all of these copies; **FFT3D**'s runtime was three orders of magnitude larger without this optimization being performed, resulting from its dataset exceeding the 512MB DRAM capacity and being paged to disk. The spill copy elimination and duplicate copy elimination optimizations were also effective for these two benchmarks, reducing the number of copies between the LS and main memory. Finally, operation hoisting increased the performance of five out of the seven benchmarks by 6–58%.

Figure 14 illustrates the effectiveness of the SPMD distribution transformation, in which the compute-heavy benchmarks scale linearly with the number of SPEs and the bandwidth-limited benchmarks scale with the available off-chip bandwidth.

	SAXPY	**SGEMV**	**FFT3D**	**SGEMM**	**CONV2D**	**GRAVITY**	**HMMER**
Total number of bulk `Copy` ops	*5*	*4*	*36*	*4*	*3*	*53*	*14*
Intra-mem. copy elim.: number of bulk `Copy` ops eliminated	0	0	18	0	0	38	4
Spill copy elim.: number of bulk `Copy` ops eliminated	0	0	0	0	0	4	0
Duplicate copy elim.: number of bulk `Copy` ops eliminated	0	0	4	0	0	0	0
Total number of ops	*86*	*154*	*412*	*186*	*124*	*731*	*280*
Operation hoisting: number of ops hoisted out of loops	0	1	6	2	1	8	8
Copy grouping: number of `Copy` grouping steps $(2 \longrightarrow 1)$	13	25	59	34	21	120	54
Operation scheduling: number of dependences added to IR	39	62	208	84	55	377	118
Total number of `Exec` ops	*5*	*18*	*59*	*6*	*8*	*18*	*16*
Exec grouping: number of `Exec` grouping steps $(2 \longrightarrow 1)$	3	16	55	5	7	11	14
Total number of loop nests	*1*	*1*	*3*	*1*	*1*	*7*	*1*
Software-pipelining: number of loops pipelined	1	1	3	1	1	7	1

Table 4. Measured counts of optimization usage

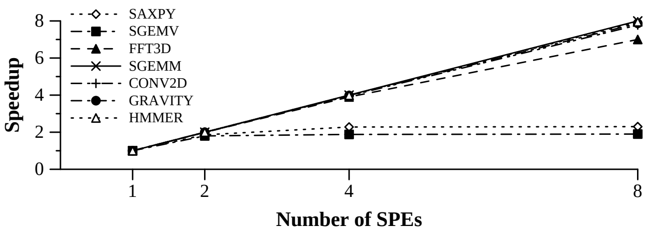

Figure 14. Performance increase due to SPMD-ization over SPEs

6.4 Compilation Time

Compilation times of our compiler are negligible compared to the time that the Cell platform compilers take to generate machine code. For example, FFT3D (one of our slowest programs to compile) takes 12s to compile on our desktop machine, 0.65s of which is spent in our compiler, with the remainder spent in XLC and GCC; by far the most time-consuming aspect of compiling a Sequoia program for Cell is the compilation of kernels by these low-level compilers. Our compiler uses greedy heuristics to solve all search problems, and iterates through the loop over the first group of optimizations in Figure 10 less than 100 times for all of our benchmarks.

7. Related Work

Our machine model is similar to the Parallel Memory Hierarchy Model (PMH) [2]: both models use trees of memories to model parallel machines. While the PMH model was used to analyze algorithm performance, we directly use a tree of memories as an *execution model* of a machine, with IR transformations being defined with respect to this execution model.

The programming system for the Imagine processor [23, 28, 13] addresses the problems of scheduling bulk operations in a two-level hierarchy and allocation of local storage. Our compiler is more general, in that it can operate on any number of levels in an explicitly managed memory hierarchy, can schedule and allocate resources at multiple scales of the application and architecture, combines both control and bulk operations in a single IR, represents and targets

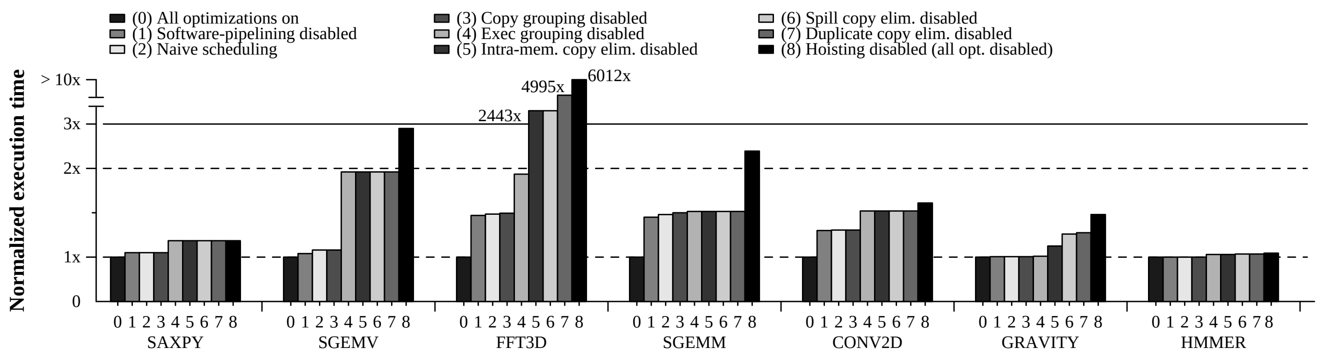

Figure 13. Normalized benchmark execution times with optimizations progressively disabled

SPMD mappings and SPMD systems, performs necessary copy-elimination transformations, and manipulates N-dimensional data objects with N-dimensional bulk transfers and complex sub-block derivations.

Eichenberger et al. presents the current status of the IBM compiler for the Cell processor [15, 14], in particular discussing code generation for Cell's PEs, including automatic vectorization and SIMD alignment. They also address some of the transformations required to utilize local stores and manage code overlays. Their implementation focuses on a two-level memory hierarchy only, and they describe a dynamic software cache approach to controlling the local stores, as opposed to our static compilation solution.

Previous work on SPMD languages has shown that optimizations for two-level hierarchies can have significant performance benefits [27]. Compilers for Titanium [36], Co-array Fortran [31], UPC [6], and ZPL [7], target SPMD style parallelism across an entire machine and focus on fine-grained "horizontal" (i.e. intra-memory-level) communication between processors. These systems recognize the importance of localization, but current implementations are for two-level hierarchies only. Optimizations for bulk transfers under limited conditions are described. The recently suggested X10 [8], Chapel [4], and Fortress [1] languages adopt similar concepts of localization and allow nested parallelism. In comparison, our compiler is structured around nested localization and parallelism, focuses on bulk operations, and inherently supports a hierarchy of operations.

A different approach to controlling software exposed memory hierarchies is to encapsulate the hierarchy in a library, as in the Message Passing Interface [30] and self-tuning, potentially cache-oblivious [18], libraries such as ATLAS [34] and FFTW [17]. An interesting hybrid compilation approach, discussed in [25] and [20], exposes library semantics to the compiler allowing for optimization and specialization across library calls.

Finally, our work manipulates bulk operations and data transfers, and differs from fine-grained techniques that address cache optimizations. Software directed prefetching (e.g., [5]) essentially breaks up a bulk transfer into smaller granularity prefetches and embeds them within arithmetic computations. Loop tiling (e.g., [35]) reorders fine-grained computation and memory accesses to reduce the number of cache misses and partition work across PEs.

8. Conclusion

We have presented the design and implementation of a compiler targeting the emerging class of compute-intensive architectures with explicitly software-managed storage hierarchies. Our focus is on the compiler's intermediate representation, which is based on the concept of a hierarchical graph of bulk operations and aggregate bulk data objects connected by dependences. We have described what we believe to be the minimal set of issues any compiler for this class of machines must address and demonstrated our solutions by compiling benchmarks for the Cell processor.

9. Acknowledgements

We would like to thank Daniel Reiter Horn for his assistance with the **HMMER** application.

References

[1] E. Allen, D. Chase, V. Luchangco, J.-W. Maessen, S. Ryu, G. Steele, and S. Tobin-Hochstadt. The Fortress language specification version 0.707. Technical report. Sun Microsystems, 2005.

[2] B. Alpern, L. Carter, and J. Ferrante. Modeling parallel computers as memory hierarchies. In *Programming Models for Massively Parallel Computers*, 1993.

[3] P. Bellens, J. M. Perez, R. M. Badia, and J. Labarta. CellSs: a programming model for the Cell BE architecture. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, 2006.

[4] D. Callahan, B. L. Chamberlain, and H. P. Zima. The Cascade high productivity language. In *Proceedings of the Ninth International Workshop on High-Level Parallel Programming Models and Supportive Environments*, pages 52–60. IEEE Computer Society, 2004.

[5] D. Callahan, K. Kennedy, and A. Porterfield. Software prefetching. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 40–52, 1991.

[6] W. W. Carlson, J. M. Draper, D. E. Culler, K. Yelick, E. Brooks, and K. Warren. Introduction to UPC and language specification. University of California-Berkeley Technical Report: CCS-TR-99-157, 1999.

[7] B. L. Chamberlain, S.-E. Choi, E. C. Lewis, C. Lin, L. Snyder, and W. D. Weathersby. The case for high level parallel programming in ZPL. *IEEE Computational Science and Engineering*, 5(3):76–86, July–September 1998.

[8] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: An object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, pages 519–538. ACM Press, 2005.

[9] A. Chow, G. Fossum, and D. Brokenshire. A programming example: Large FFT on the Cell Broadband Engine. http://www-306.ibm.com/chips/techlib/techlib.nsf/techdocs/0AA2394A505EF0FB872570AB005BF0F1, 2005.

[10] L. Cico, R. Cooper, and J. Greene. Performance and programmability of the IBM/Sony/Toshiba Cell Broadband Engine processor. In *Workshop on Edge Computing Using New Commodity Architectures (EDGE)*, 2006.

[11] ClearSpeed. CSX600 Processor Datasheet. http://www.clearspeed.com/, 2005.

[12] W. J. Dally, P. Hanrahan, M. Erez, T. J. Knight, F. Labonte, J. Ahn, N. Jayasena, U. J. Kapasi, A. Das, J. Gummaraju, and I. Buck.

Merrimac: Supercomputing with streams. In *Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*, page 35, 2003.

[13] A. Das, W. J. Dally, and P. Mattson. Compiling for stream processing. In *Proceedings of the 2006 International Conference on Parallel Architectures and Compilation Techniques*, 2006.

[14] A. Eichenberger, J. O'Brien, K. O'Brien, P. Wu, T. Chen, P. Oden, D. Prener, J. Shepherd, B. So, Z. Sura, A. Wang, T. Zhang, P. Zhao, M. Gschwind, R. Archambault, Y. Gao, and R. Koo. Using advanced compiler technology to exploit the performance of the Cell Broadband Engine architecture. *IBM System Journal*, 45(1), 2006.

[15] A. Eichenberger, K. O'Brien, K. O'Brien, P. Wu, T. Chen, P. Oden, D. Prener, J. Shepherd, B. So, Z. Sura, A. Wang, T. Zhang, P. Zhao, and M. Gschwind. Optimizing compiler for the Cell processor. In *Proceedings of the 2005 International Conference on Parallel Architectures and Compilation Techniques*, September 2005.

[16] K. Fatahalian, T. J. Knight, M. Houston, M. Erez, D. R. Horn, L. Leem, J. Y. Park, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan. Sequoia: Programming the memory hierarchy. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, 2006.

[17] M. Frigo. A fast Fourier transform compiler. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, volume 34, pages 169–180, May 1999.

[18] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, page 285, 1999.

[19] T. Fukushige, J. Makino, and A. Kawai. GRAPE-6A: A Single-Card GRAPE-6 for Parallel PC-GRAPE Cluster Systems. *Publications of the Astronomical Society of Japan*, 57:1009–1021, dec 2005.

[20] S. Z. Guyer and C. Lin. An annotation language for optimizing software libraries. In *Second Conference on Domain-Specific Languages*, pages 39–52, October 1999.

[21] D. R. Horn, M. Houston, and P. Hanrahan. ClawHMMER: A streaming HMMer-search implementation. In *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, page 11, Washington, DC, USA, 2005. IEEE Computer Society.

[22] IBM. Cell Broadband Engine Architecture Version 1.01. http://www-306.ibm.com/chips/techlib/techlib.nsf/techdocs/1AEEE1270EA277 6387257060006E61BA, August 8 2005.

[23] U. J. Kapasi, P. Mattson, W. J. Dally, J. D. Owens, and B. Towles. Stream scheduling. In *Proceedings of the 3rd Workshop on Media and Streaming Processors*, pages 101–106, 2001.

[24] U. J. Kapasi, S. Rixner, W. J. Dally, B. Khailany, J. H. Ahn, P. Mattson, and J. D. Owens. Programmable stream processors. *IEEE Computer*, August 2003.

[25] K. Kennedy, B. Broom, K. Cooper, J. Dongarra, R. Fowler, D. Gannon, L. Johnsson, J. Mellor-Crummey, and L. Torczon. Telescoping Languages: A Strategy for Automatic Generation of Scientific Problem-Solving Systems from Annotated Libraries. *Journal of Parallel Distributed Computing*, 61:1803–1826, December 2001.

[26] M. S. Lam. *A Systolic Array Optimizing Compiler*. Kluwer Academic Publishers, 1989.

[27] B. Liblit and A. Aiken. Type systems for distributed data structures. In *Symposium on Principles of Programming Languages*, 2000.

[28] P. Mattson. *A Programming System for the Imagine Media Processor*. PhD thesis, Stanford University, 2002.

[29] S. McPeak and D. Wilderson. Elsa: The Elkhound-based C/C++ Parser. http://www.cs.berkeley.edu/˜smcpeak/elkhound, 2005.

[30] Message Passing Interface Forum. MPI: A Message Passing Interface Standard, May 1994.

[31] R. W. Numrich and J. Reid. Co-array Fortran for parallel programming. *SIGPLAN Fortran Forum*, 17(2):1–31, 1998.

[32] D. Pham, S. Asano, M. Bolliger, M. N. Day, H. P. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Riley, D. Shippy, D. Stasiak, M. Suzuoki, M. Wang, J. Warnock, S. Weitzel, D. Wendel, T. Yamazaki, and K. Yazawa. The design and implementation of a first-generation CELL processor. In *IEEE International Solid-State Circuits Conference*, 2005.

[33] RapidMind. http://rapidmind.net/.

[34] R. C. Whaley and J. J. Dongarra. Automatically tuned linear algebra software. In *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing (CDROM)*, 1998.

[35] M. Wolfe. More iteration space tiling. In *Proceedings of the 1989 ACM/IEEE Conference on Supercomputing*, pages 655–664, 1989.

[36] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: A high-performance Java dialect. In *ACM 1998 Workshop on Java for High-Performance Network Computing*, 1998.