The Multicluster Architecture: Reducing Cycle Time Through Partitioning[*]

Keith I. Farkas[†] farkas@pa.dec.com
Paul Chow[‡] pc@eecg.toronto.edu
Norman P. Jouppi[†] jouppi@pa.dec.com
Zvonko Vranesic[‡] zvonko@eecg.toronto.edu

[†]Digital Equipment Corporation
Western Research Lab
250 University Avenue
Palo Alto, California 94301

[‡]Electrical and Computer Engineering
University of Toronto
10 Kings College Road
Toronto, Ontario, Canada
M5S 3G4

Abstract

The multicluster architecture that we introduce offers a decentralized, dynamically-scheduled architecture, in which the register files, dispatch queue, and functional units of the architecture are distributed across multiple clusters, and each cluster is assigned a subset of the architectural registers. The motivation for the multicluster architecture is to reduce the clock cycle time, relative to a single-cluster architecture with the same number of hardware resources, by reducing the size and complexity of components on critical timing paths. Resource partitioning, however, introduces instruction-execution overhead and may reduce the number of concurrently executing instructions. To counter these two negative by-products of partitioning, we developed a static instruction scheduling algorithm. We describe this algorithm, and using trace-driven simulations of SPEC92 benchmarks, evaluate its effectiveness. This evaluation indicates that for the configurations considered, the multicluster architecture may have significant performance advantages at feature sizes below 0.35μm, and warrants further investigation.

1 Introduction

A continuing challenge in the design of microprocessors is the need to balance the complexity of the hardware against the speed at which it can be clocked. This challenge exists because increased hardware complexity can impact the cycle time of a processor in two ways. First, if a component is on a critical timing path, increasing the complexity of the component may increase its cycle time, and thus, that of the processor. Second, because complex components may be physically larger and further apart, the time for signals to travel between them may be greater, thereby necessitating an increase in the processor's cycle time. One approach to reducing these two consequences of complexity is to partition the hardware, thereby reducing the complexity and size of components.

In this paper, we introduce a dynamically-scheduled, partitioned architecture called the *multicluster architecture*. This architecture implements dynamic scheduling using dispatch queues and explicit register renaming hardware, a basis that is used in the DEC Alpha 21264 [1] and the MIPS R10000 [2]. *Dispatch queues* are used to maintain the pool of instructions from which the instruction scheduler issues instructions to the functional units, while register renaming is used to map the *architectural registers* (i.e., those named by instructions) to a larger set of *physical registers*.

In the multicluster architecture, the dispatch queues, register files, and functional units are distributed across multiple clusters. The instructions that are executed by a cluster C are those dispatched from the dispatch queue of cluster C, and these instructions are the only instructions that can read or write the physical registers of cluster C. Each cluster is assigned a subset of the architectural registers. This assignment along with the architectural registers named by an instruction determines the cluster(s) that execute the instruction. Multiple-cluster execution is used whenever an instruction either names source registers that are not accessible from within one cluster or names a destination register that is not uniquely assigned to one cluster (Section 2.1 discusses the assignment of architectural registers to clusters). The instructions for all clusters are obtained from a single, shared stream of instructions that are fetched from a single instruction cache. The data cache is also shared by all clusters.

The isolation of each cluster's components provides two benefits relative to a processor with a non-partitioned architecture that can issue the same number of instructions per cycle as all the clusters of a multicluster processor. First, because each cluster issues fewer instructions per cycle, the register files of a given cluster require fewer read/write

[*]Copyright 1997 IEEE. Published in the Proceedings of Micro-30, December 1-3, 1997 in Research Triangle Park, North Carolina. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works, must be obtained from the IEEE. Contact: Manager, Copyrights and Permissions / IEEE Service Center / 445 Hoes Lane / P.O. Box 1331 / Piscataway, NJ 08855-1331, USA. Telephone: + Intl. 908-562-3966.

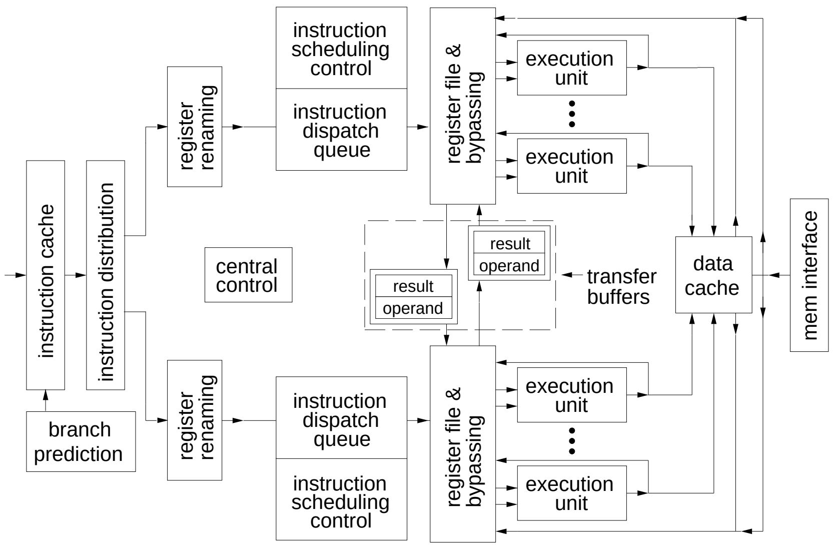

Figure 1: A dual-cluster processor built within the multicluster architecture framework.

ports. As the number of read/write ports largely determines the cycle time of a register file, partitioning reduces its cycle time and perhaps that of the processor. Indeed, the DEC Alpha 21264 has a partitioned integer register file because the integer register file is on a critical timing path [1]. Second, because each cluster issues fewer instructions per cycle, the dispatch queue of a given cluster not only requires fewer read/write ports, but also requires less complex instruction-scheduling logic. Consequently, the cycle time of the instruction-scheduling hardware will likely be smaller. For example, since the instruction queues of the MIPS R10000 are on a critical timing path [2], a smaller cycle time might have been obtained had the queues been partitioned.

We begin the discussion of the multicluster architecture in the next section by describing the architecture, and the process by which instructions are executed. Then, in Section 3, we describe the most successful of the static instruction scheduling algorithms we developed for it. In Section 4, we then present the results of a simulation-based evaluation of this algorithm. Finally, in Section 5, we reexamine the motivation for the architecture in light of the results, and suggest areas for future work.

Without loss of generality, we discuss the multicluster architecture in terms of a multicluster processor with two clusters. Further, in our model of such a processor (Figure 1), each cluster comprises a single dispatch queue (rather than the multiple queues used in the R10000 and Alpha 21264), and two register files, one for integer values, and the second for floating-point values.

2 The Architecture

This section describes the multicluster architecture, the process by which instructions are executed, and tradeoffs in the design of a multicluster processor. A more complete description is presented in [3].

2.1 Instruction Distribution and Execution

Instructions are read from the instruction cache in fetch order and are distributed, also in fetch order, to one or both clusters. If an instruction can't be distributed to a cluster because a dispatch-queue entry or a physical register is not available, the instruction stream is stalled until the required resource becomes available. The distribution of instructions to the clusters is based on the registers named by each instruction and the cluster(s) to which the architectural registers have been assigned. We use the term *local register* to refer to an architectural register that has been assigned to one cluster, and the term *global register* to refer to an architectural register that has been assigned to both clusters. Global registers would typically be used for stack and global pointers, as well as other commonly used variables. Owing to the ease of detecting the architectural registers named by an instruction and the cluster(s) to which each register has been assigned, the hardware required for instruction distribution is relatively simple. In the simplest case, for a two-cluster configuration, local registers could be identified using a bit-mask, and the cluster assignment for these registers could be based on whether the register number is even or odd. Simple distribution criterion are especially important if distribution to more than one cluster is anticipated. Although a simple hardware mechanism exists to support the dynamic reassignment of the architectural registers (see [3]), we assume that the assignment is static.

When an instruction is executed that names a local register R as a destination, the value computed by the instruction is stored in a physical register of the cluster to which register R has been assigned. But, when an instruction is executed that names a global register G as a destination, the value computed by the instruction is stored in a physical register of each cluster. Thus, two physical registers are required to maintain the value of a global register, one in each cluster, but only one physical register is required to maintain the value of a local register.

Execution Details

The execution of an instruction requires that the hardware perform a sequence of steps, with this sequence dependent on the cluster(s) to which each of the named (architectural) registers are assigned. There are various scenarios under which these sequences occur, and these can be grouped into five main categories. To better examine these categories, consider the the integer add instruction $r_2 \leftarrow r_0 + r_1$, and a scenario from each category.

Scenario one: suppose that the three registers named by this instruction are local registers assigned to cluster C_1. To perform the add, the hardware first distributes the instruction to cluster C_1. The distribution process comprises three tasks: the source registers r_0 and r_1 are mapped to the physical registers of cluster C_1, the destination register r_2 is renamed to a free physical register of cluster C_1, and the instruction is inserted into a free entry in the dispatch queue of cluster C_1. Later, after the source operands are available and when a suitable functional unit is available, the hardware issues the instruction, reads its source operands, performs the add, and writes the result into the bound physical register.

Scenario two: suppose again that all three registers are local registers, but with source register r_1 and destination register r_2 assigned to cluster C_1, and source register r_0 assigned to cluster C_2. To perform the add, the hardware distributes a copy of the instruction to both clusters. *Dual distribution* provides the mechanism by which the source operand that is not available in cluster C_1 (i.e., r_0) is transfered to the cluster that performs the computation (i.e., cluster C_1).

The *master copy* does the computation with the *slave copy* supplying one of the source operands. The master copy is executed by cluster C_1 because the majority of the local registers named by the instructions are assigned to cluster C_1 (the selection of the master copy's cluster is discussed further in [3]). When the master copy is distributed to cluster C_1, r_2 is renamed using a physical register belonging to this cluster. However, when the slave copy is distributed to cluster C_2, it is not allocated a physical register because the destination of the instruction is a local register that has been assigned to cluster C_1. Figure 2 shows the steps taken to execute the add instruction by executing the master and slave copies. These steps are explained below.

There exists a data dependence between the slave copy

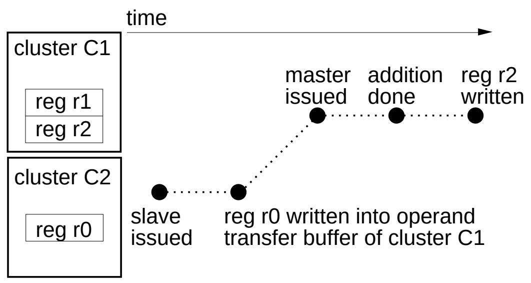

Figure 2: Dual execution of the instruction $r_2 \leftarrow r_0 + r_1$ when operand r_0 is forwarded to cluster C_1.

and the immediately preceding instruction in fetch order that wrote r_0. The slave copy can be issued only after this dependence is resolved and when an integer issue slot is available. The hardware requires an integer issue slot to issue the slave copy because the slave copy must read the value of r_0 from the integer register file, and to do so requires access to a read port. There also exists a data dependence between the master copy and the immediately preceding instruction in fetch order that wrote r_1, and a dependence between the master and slave copies. This inter-copy dependence guarantees that the master copy will be issued only when its second operand, that is r_0, is available. Therefore, the master copy will be issued after both input dependences are resolved and when a suitable functional unit is available.

In the write-back stage of the execution pipeline, the slave copy writes the value of r_0 into an entry in the *operand transfer buffer* (Figure 1) of cluster C_1, the master copy's cluster. An entry in the operand transfer buffer is allocated to the slave copy when it is issued. If an entry is not available, then the slave copy is blocked from being issued, and in certain circumstances, an instruction-replay exception is required to avoid issue deadlock (see [3] for more details). When the slave copy writes the value into the allocated entry, it also stores the unique ID of the instruction of which it is a copy. This ID is used by the hardware to associatively search the operand transfer buffer to locate the operand for the associated master copy. The dependence between the master copy and the slave copy is removed when the slave copy is issued, thereby permitting the master copy to be issued as soon as the next cycle. After the master copy obtains the value, the entry allocated to the slave copy is freed. This entry can be used by another instruction in the next cycle.

Scenario three: suppose again that the two source registers are local registers assigned to cluster C_1, but that the destination register r_2 is assigned to cluster C_2. To perform the add, the hardware again distributes a copy of the instruction to both clusters, but this time, dual distribution provides the mechanism by which the result of the computation is transfered to the cluster to which the destin-

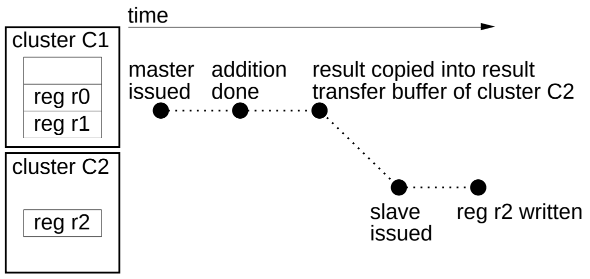

Figure 3: Dual execution of the instruction $r_2 \leftarrow r_0 + r_1$ when result is forwarded to cluster C_2.

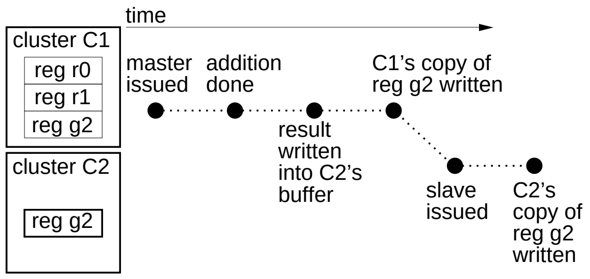

Figure 4: Dual execution of the instruction $r_2 \leftarrow r_0 + r_1$ when global result is forwarded to cluster C_2.

ation register has been assigned. Figure 3 shows the steps taken to execute the add instruction for this scenario; these steps are explained below.

Because the two source operands for the operation are located on the same cluster, the master copy of the instruction will be issued first. Then, the result is computed, and forwarded to the slave copy. The slave copy is then issued and it writes the forwarded result into the physical register bound to r_2. Unlike in the second scenario, the physical register is allocated to the slave copy because the destination register is assigned to the slave copy's cluster C_2 and not the master copy's cluster C_1. The data transfer is performed by writing the value computed by the master copy into an entry in the *result transfer buffer* of cluster C_2. This entry is freed after the slave copy reads the result out of the entry prior to writing the result into the bound physical register. To prevent the slave copy from being issued before the result is available, there exists a dependence between the slave and master copies. This dependence is removed two cycles before the master copy is due to finish computing the result, and, thus, for simple one-cycle latency instructions like the add, the slave copy can be issued as soon as one cycle after the master copy is issued (see [3] for a description of the execution pipeline). Finally, as also discussed in [3], separate result and operand transfer buffers are provided to reduce implementation complexity and to reduce the number of times an instruction-replay exception is required to free up a buffer entry.

Scenario four: suppose again that both source registers are local and assigned to cluster C_1, but that the destination register r_2 is a global register. Because the destination is a global register, both the slave and master copies are allocated physical registers when they are inserted into their respective dispatch queues. Thus, dual distribution provides the mechanism by which (1) a physical register in each cluster is allocated for the new value of r_2, (2) the inter-instruction dependences arising from the use of r_2 are maintained in each dispatch queue, and (3) the value computed by the master copy is written into the two allocated physical registers.

The same sequence of steps as for the third scenario is performed to execute the add, with the exception that, when the master copy writes the result into the slave copy's result transfer buffer, the master copy *also writes the result into the physical register it was allocated*. Figure 4 illustrates this scenario.

Scenario five: finally, suppose again that source registers r_0 and r_1 are local registers and the destination register r_2 is a global register, but that r_0 has been assigned to cluster C_2, while r$_1$ has been assigned to cluster C_1. As in the second scenario, the slave copy is issued only after its data dependence is resolved and when both an integer issue slot and an operand transfer buffer entry are available. Once the slave copy writes the operand into the allocated entry, the hardware suspends it. The master copy is then issued only after its data dependence is resolved, and when both a suitable functional unit and a result transfer buffer entry are available. When the master copy obtains the forwarded source operand, it frees the operand transfer buffer entry (which is associated with its cluster). It then computes the result, and writes it into both the allocated result transfer buffer entry (which is associated with the slave copy's cluster) and into its own register file. Finally, the slave copy is awakened, it obtains the result, frees the result transfer buffer entry, and writes the result into its own register file. Figure 5 illustrates the steps for this scenario.

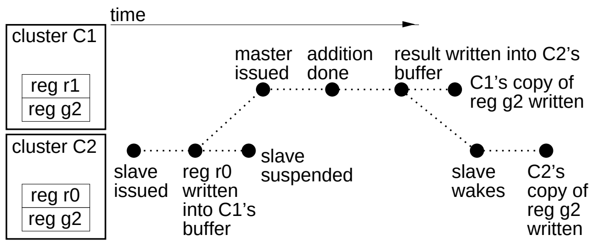
time
cluster C1
reg r1
reg g2
cluster C2
reg r0
reg g2
master issued
addition done
result written into C2's buffer
C1's copy of reg g2 written
slave issued
reg r0 written into C1's buffer
slave suspended
slave wakes
C2's copy of reg g2 written

Figure 5: Dual execution of the instruction $r_2 \leftarrow r_0 + r_1$ when operand r_0 is forwarded to cluster C_1 and global result is forwarded to cluster C_2.

The performance obtained from a multicluster processor is affected by the number of instructions distributed to one cluster, the number distributed to two clusters, and how these instructions are arranged in the instruction fetch stream; this order is important because it affects resource availability. Dual-distributed instructions require more hardware resources than instructions distributed to one cluster. Thus, a larger number of dual-distributed instructions contributes to a smaller instruction throughput. In addition, when an instruction is dual distributed, the two copies operate as a pair to perform the required task. Because the mechanism that supports this cooperation introduces some overhead, a dual-distributed instruction requires more clock cycles to execute than a single-distributed instruction (assuming all other conditions are the same). Thus, not only do dual-distributed instructions contribute to a reduction in throughput, they may also require more clock cycles to execute. However, note that these two negative effects are offset by the reduction in the cycle time of the processor clock that is provided by partitioning the hardware resources. Consequently, even though an application may require more clock cycles to execute on a multicluster processor than on a single-cluster processor, the run time may be reduced.

2.2 Related Architectures

A number of similarities and differences exist between proposed and existing architectures and the multicluster architecture. This section briefly examines a few such architectures.

The aim of the multicluster architecture is to decrease the cycle time of the processor to permit a single thread of execution to run faster. Partitioning of components to increase performance was also one of the aims of the decoupled access/execute architecture proposed by Smith [4]. This architecture and the multicluster architecture both consists of two tightly-coupled clusters interconnected by buffers. However, the two clusters of the decoupled access/execute architecture are statically scheduled, with one responsible for reading and writing memory, and the other responsible for computing results. The decoupled access/execute architecture also requires that values be written into and read from the inter-cluster buffers in the same order. Thus, while the access and execute instruction streams may “slip” with respect to each other, instructions from different streams cannot be executed out of order.

In the multicluster architecture and the Multiflow architecture [5], the physical registers and the functional units of both architectures are distributed among the clusters of the respective machine. However, while a mechanism exists in the multicluster architecture for accessing the registers in other clusters, in the Multiflow architecture, for an ALU (arithmetic logic unit) A to use a value stored in the registers associated with another ALU, this value must first be explicitly copied to a register in the register file of A. This two-step process is coordinated by the compiler, a fact that underlines that the hardware is completely predictable, which is not true for the multicluster architecture due to its use of dynamic scheduling. One implication of the predictability of the Multiflow architecture is that it allows the Multiflow compiler to balance the work to be performed across all ALUs and to encode it into a single instruction stream. However, as discussed in Section 3, balancing the workload across the clusters of the multicluster architecture is more difficult due to the unpredictability of dynamic scheduling.

The Multiscalar architecture [6] is in some ways similar to the the multicluster architecture in that both derive benefits from partitioning a large processor into several clusters of execution resources. In addition, both architectures share the common need for good static scheduling of the application to keep all functional units busy. However, there are a number of important differences. First, the basis used to distribute the instructions to the clusters of a Multiscalar architecture is information encoded in the binary by the compiler, whereas for the multicluster architecture, the basis is the architectural registers named by each instruction. Second, each cluster of the Multiscalar architecture independently fetches the instructions assigned to it, whereas the clusters of a multicluster architecture share a common instruction fetch stream. As a result of these two differences, instruction scheduling for the multicluster architecture is more complex. A third important difference is that, while the Multiscalar architecture attempts to exploit parallelism between threads each consisting of many basic blocks, while the multicluster architecture primarily exploits parallelism within and between basic blocks.

Simultaneously multithreaded processors [7] and tightly-coupled multiprocessors [8] share the property that they are capable of simultaneously executing independent or co-operating sequences of instructions, called threads. The aim of supporting this capability is to reduce the number of clock cycles required to execute multiple threads sequentially. In contrast, the aim of the multicluster architecture is to decrease the cycle time of the processor to permit a single thread of execution to run faster.

3 Static Instruction Scheduling

Static instruction scheduling is the process by which, prior to the execution of an application, the machine-level instructions are ordered with the goal of minimizing the number of clock cycles required to execute the application. For a multicluster processor to meet this goal, during the execution of an application, the instructions must be balanced across the clusters, with each cluster concurrently performing similar amounts of work, and with no instructions executed by more than one cluster. A balanced workload is desired because each cluster contains only a subset of the total available hardware resources, while single-cluster instruction distribution is desired because dual distribution increases hardware-resource requirements and execution latency. However, the workload balance cannot be directly addressed by the compiler because the work done by a cluster is a function of the order in which instructions are issued, and the issue order is not deterministic for dynamically-scheduled processors. Thus, the compiler can only indirectly address the workload balance by seeking to balance the dynamic distribution of instructions, under the assumption that a balanced distribution will result in a balanced workload.

The objective of a balanced distribution competes against the objective of a minimum number of dual-distributed instructions. However, in practice, for the benchmarks considered, the performance cost of dual-distributing instructions was much less than the cost of not taking full advantage of the available hardware resources (see [3]). Consequently, the primary objective is to generate a code schedule, which when run, generates an instruction stream in which the instruction-distribution is balanced. To address the two objectives, the compiler considers each (static) instruction individually, and seeks a register assignment for the data values used by the instructions. The criterion used to select a register and hence a cluster for a data value is whether the instruction distribution is likely to be balanced. If it is not, the compiler must select a register that is assigned to the under-subscribed cluster. If the distribution is likely balanced, the compiler can select a register that allows the instruction to be distributed to only one cluster.

To select a register for the value used by an instruction, the compiler must first determine whether the instruction distribution is likely balanced around the instruction at run time. To do so, the compiler must know the order in which instructions will be fetched and distributed. Because this information is implicit in an ordered sequence of instructions, the allocation of values to registers must be carried out after the instructions are ordered into a code schedule. That is, *prepass scheduling* must be used. Once the instruction balance has been estimated for an instruction, to make the register selection(s) for the instruction, the compiler must take into account the inter-dependences between instructions that arise from one or more instructions using a value generated by another. As a result of this source of dependences, decisions made for one instruction can affect those made for other instructions. A useful abstraction for capturing this source of dependences is that of a *live range* [9].

3.1 Code Generation Methodology

The code generation methodology, which takes into account the issues introduced in the previous subsection, comprises the following six steps.

1. The application is compiled into an intermediate language (IL) to which are applied conventional optimizations like common subexpression elimination and constant propagation.

2. The IL instructions are arranged into a (static) code schedule. The IL instructions correspond one-to-one to the machine-level instructions of the processor, but unlike the machine-level instructions, the IL instructions name live ranges and not registers.

3. The live ranges associated with the stack pointer and the global pointer are designated as candidates for global registers; all other live ranges are designated as candidates for local registers. (The rational for this designation is discussed in [3].)

4. The live ranges that are candidates for local registers are then partitioned with the goal of maximizing the concurrent utilization of both clusters, and minimizing the number of dual-distributed instructions.

5. The live ranges are allocated to the architectural registers with global-register candidates allocated to global registers and local-register candidates allocated to local registers.

6. The machine-level instructions (including those required for register spilling) are arranged into a code schedule.

Steps 1, 2, 4, and 5 correspond to the four compilation problems: *code optimization*, *code scheduling*, *live range partitioning*, and *register allocation*. Although solutions to these four problems must handle the unique characteristics of the multicluster architecture, we have focused on solving the third problem, live range partitioning, since this problem captures most of the idiosyncrasies of the architecture. In Section 3.5, we briefly describe the most successful of the novel techniques we developed for solving this problem, while in Sections 3.2-3.4, we briefly describe how we modified existing techniques to solve the other three compilation problems. A more extensive description of our solutions to the four problems is given in [3].

3.2 Code Optimization

The code optimization problem can be solved using techniques such as those described by Aho et al. [9]. Since this problem arises early in the compilation of an application, solutions to it are relatively independent of the unique characteristics of the multicluster architecture. Thus, to limit the scope of our research, the existing techniques were used without modification.

3.3 Code Scheduling

The code scheduling problem can best solved using techniques that tend to generate large basic blocks. Such techniques, like trace scheduling [5], are preferable due to the necessity of estimating the run-time instruction imbalance on a per-basic-block basis when performing live range partitioning. The run-time instruction imbalance is a function of the order in which the basic blocks appear in the fetch stream, and the static imbalance of each block. To ensure a given degree of balance is obtained at run time, the compiler would have to take into account all control flow paths by which a given basic block can be reached, and the cluster allocation of the live ranges used within the basic block. Owing to the complexity of concurrently considering these two effects, scheduling on a per-basic-block basis is mandated.

3.4 Register Allocation

Finally, the register allocation problem can best be solved using the graph-coloring technique developed by Briggs et al. [10]. This technique is most suitable because it separates the process of coloring nodes from the process of spilling live ranges. The separation of these two phases provides a convenient framework for implementing the desire to spill a live range first to a local register in the other cluster and, if no register is available, then to memory. In addition, the separation of the phases increases the likelihood that a live range will be allocated a register [10] and allows for other optimizations [3]. These features compete against the increased likelihood that more live ranges will be spilled since each cluster of the multicluster architecture is allocated only a subset of the architectural registers.

3.5 Live Range Partitioning

This section describes the *local scheduler*, which was developed to solve the live range partitioning problem. The approach taken by the local scheduler is to determine for each live range L in the intermediate-language representation of an application, the cluster to which L should be assigned so as to ensure the instruction-distribution at run time is balanced in the vicinity of every instruction that reads or writes L.

To determine cluster assignments for the live ranges, the local scheduler begins by sorting the basic blocks according to the number of times the first instruction in each basic block is estimated to be executed[1]. Basic blocks with equal estimates are sorted by the number of static instructions in each block. Then, the basic block having the largest estimate and the greatest number of instructions is removed from the list, and a bottom-up, in-order traversal is carried out on the instructions in the block. As an example, consider the control flow graph shown in Figure 6, and the execution

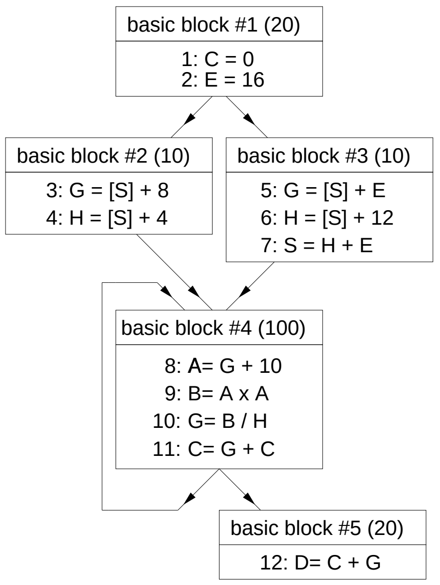

Figure 6: Example control flow graph. In the graph, the numbers in parentheses give the dynamic-execution estimates for each basic block. Furthermore, while live range S is assumed to be a candidate for a global register, all other live ranges are assumed to be candidates for local registers.

estimates for each block given by the numbers in the parentheses. For this control flow graph, the basic blocks will be traversed in the order 4, 1, 5, 3, and 2.

The purpose of this traversal is to visit each instruction in turn, and if the instruction writes an unassigned live range, choose a cluster for the live range. A cluster is chosen for a live range L, which is written by an instruction I, by first examining the instruction-distribution around the instruction. If the distribution is not balanced, the cluster chosen for L will be the one that reduces the degree of imbalance. An instruction-distribution is considered unbalanced in the vicinity of an instruction I if, at run time, at the point in time that I is distributed to one or both clusters for execution, there has been more than a given number of instructions distributed to one cluster than the other; this number is a compile-time constant. (The reader is referred to [3] for a more formal definition of imbalance.) If the distribution is estimated to be balanced, however, then the scheduler determines the cluster that is preferred by the majority of the instructions that read or write L. The scheduler selects cluster C as the preferred cluster for one of these instructions if the assignment of L to C will allow the instruction to be distributed to only one cluster.

Thus, once the bottom-up traversal of a basic block is completed, there will remain no unassigned live ranges among those that are written by the instructions in the basic block. After completing the bottom-up traversal of a basic

[1]These estimates are derived from profiling the execution of the application on a dual-cluster processor.

block, the scheduler removes the next basic block from the list, and carries out a bottom-up traversal on its instructions. This process continues until all basic blocks are visited. As a result of this process, the cluster-assignment for a live range will be determined the first time an instruction is encountered that writes it during the basic-blocks traversals. Thus, for the example of Figure 6, the local scheduler will visit the basic blocks in the order 4, 1, 5, 3, and 2, and, as a result, the live ranges will be assigned to clusters in the order C, G, B, A, E, D, and H. Live range S, however, is not considered during live range partitioning because it is assumed to be a candidate for a global register.

4 Performance Assessment

The performance impact of the schedulers we developed was determined by using ATOM [11], an object-code instrumentation system, to simulate the execution of several SPEC92 benchmarks. For these simulations, we first compiled the benchmarks using the standard Digital Unix compilers for 21064-based workstations to produce a *native binary*. Then, using ATOM, we analyzed the native binary to discover the data and control dependences between instructions, and the live ranges these instructions read and write. While the use of ATOM and the standard compilers prohibited the use of compilation techniques (e.g., loop unrolling) not supported by the Digital Unix compilers, but which might improve the performance of the multicluster architecture, it simplified the implementation of the schedulers and permitted the fundamental scheduling problems to be the focus of the work.

After identifying the live ranges, they were partitioned and assigned to the architectural registers using one of the schedulers; the object code generated by the schedulers is called the *rescheduled binary*. For this step, the schedulers assumed that the even-numbered architectural registers were assigned to cluster C_1 and the odd-numbered registers to cluster C_2. This architectural-register-to-cluster assignment was determined through the analysis of early simulation results (see [3] for more details). Then, the rescheduled binary was instrumented using ATOM and linked with the multicluster simulator. Instrumentation took into account the new register assignments for each instruction and any code required to spill live ranges. Finally, the combined application-simulator was then run and the number of (simulated) clock cycles required to execute the application was recorded. This number is our performance metric.

To evaluate the impact of rescheduling, we compared the performance obtained from the rescheduled binaries when they were executed on a dual-cluster processor to that obtained when the native binary was executed on a single-cluster processor. For this comparison, the single-cluster processor was configured with the same number of resources as the entire dual-cluster processor. Although the evaluation was done for both four-way and eight-way issue processors, the results presented here are only for an eight-way issue processor because these more clearly show the important trends.

4.1 Simulation Model

The single-cluster and dual-cluster processors each implement a RISC, superscalar processor whose instruction set is based on the DEC Alpha instruction set. Each processor supports non-blocking loads and non-blocking stores, and allows all instructions to be speculatively executed. Each processor includes separate data and instruction caches, each of which is a 64-Kbyte, two-way set associative cache. The data cache is assumed to use an inverted MSHR [12], and thus, imposes no restriction on the number of in-flight cache misses. The memory interface between the instruction and data caches, and the lower levels of the memory hierarchy is assumed to have a 16-cycle fetch latency and unlimited bandwidth.

In each clock cycle, each processor can fetch up to 12 instructions from the instruction cache, and can insert these instructions into the dispatch queue(s). The single-cluster processor has a 128-entry dispatch queue, while each cluster of the dual-cluster processor has a 64-entry dispatch queue. To enable fetching beyond conditional branches, both processors use a branch prediction scheme proposed by McFarling [13] that comprises a bimodal predictor, a global history predictor, and a mechanism to select between them; all other control flow instructions are assumed to be 100% predictable. As instructions are inserted into a dispatch queue, the architectural registers named by each are renamed to the corresponding physical registers. The single-cluster processor has 128 integer and 128 floating point registers, while each cluster of the dual-cluster processor has 64 integer and 64 floating point registers.

In each clock cycle, the instruction scheduling logic selects instructions to issue out of the dispatch queues using a greedy algorithm that issues the oldest, ready-to-issue instruction in a queue first. The single-cluster processor can issue up to eight instructions per cycle, while each cluster of the dual-cluster processor can issue at most four instructions per cycle. The instruction issue rules for the processors are given in the first and second rows of Table 1, while the functional unit latencies are given in the third row. Correctly executed instructions are retired in program order with each processor capable of retiring no more than eight per cycle. Finally, both processors are assumed to implement precise exceptions, and each cluster of the dual-cluster processor has eight operand- and eight result-buffer entries.

4.2 Results

A common (and expected) trend is that more clock cycles are required to execute a benchmark on the dual-cluster processor than on the single-cluster processor. This increase is attributable to an increase in the number of stalls

#			instruction types								
			all	integer			floating point			loads &	control
				all	multiply	other	all	divide	other	stores	flow
1	number issued	single	8	8	8	8	4	4	4	4	4
2	per cycle	dual, per cluster	4	4	4	4	2	2	2	2	2
3	latency in cycles				6	1		8/16	3	1†	1

Table 1: Instruction-issue rules for the single-cluster (row #1) and the dual-cluster (row #2) processors. Row #3 gives the functional-unit latencies. All functional units are fully pipelined with the exception of the floating-point divider. The divider is not pipelined and has an eight-cycle latency for 32-bit divides, and a 16-cycle latency for 64-bit divides. †There is a single load-delay slot.

of the instruction-fetch stream, a reduction in the number of instructions issued per cycle, an increase in the instruction-issue disorder, and a slight increase in the data-cache miss rate due to the increased issue disorder. A useful metric is $\frac{N_{dual}}{N_{single}}$, where N_{single} is the number of (simulated) clock cycles required to execute the native binary of a benchmark on the single-cluster processor, while N_{dual} is the number of (simulated) clock cycles required to execute either the native binary or the rescheduled binary on the dual-cluster processor. This performance ratio is said to indicate a *speedup* if it is less than one, and a *slowdown* if it is greater than one. Note, because performance is really a product of the number of clock cycles and the period of the clock T, a multicluster processor will perform as well as or better than a single-cluster processor if the clock period of the multicluster processor T_{dual} is less than or equal to $\frac{N_{single}}{N_{dual}} \times T_{single}$.

The performance ratios for each benchmark are presented in Table 2 as the percentage speedup/slowdown. This percentage is equal to $100(1 - \frac{N_{dual}}{N_{single}})$. Comparison of all 12 data points indicates that in general, the benchmarks incurred a slowdown in the number of clock cycles of between 5% and 41%. Further comparison of the performance ratios indicates that with the exception or *ora*, the use of the local scheduler significantly reduces the slowdown incurred from running on the dual-cluster processor. Furthermore, in the case of *compress*, with the use of the local scheduler, the benchmark performs better on the dual-cluster processor than the single-cluster processor. This increase in performance is due to the single-cluster processor having a larger dispatch queue. The size of the dispatch queue is important for two reasons. First, with larger dispatch queues, there is likely to be a greater amount of time between when a branch prediction is made and when the branch predictor tables are updated with the direction taken by the branch[2]. Hence, with larger dispatch queues, a greater number of predictions may be based on information that may not reflect the direction taken by immediately preceding branches in program order. The size of the dispatch queue is also important because a larger dispatch queue allows for more disorder in the issuing of instructions. In the case of *compress*, this increase in issue disorder leads to an increase in the cache miss rate, and thus, a performance degradation.

benchmark (1)	speedup ratio	
	none (2)	local (3)
compress	-14	+6
doduc	-21	-15
gcc1	-15	-10
ora	-5	-22
su2cor	-36	-25
tomcatv	-41	-19

Table 2: The speedup ratios $100(1 - \frac{N_{dual}}{N_{single}})$ obtained when the benchmarks were not rescheduled (column 2) and when the local scheduler was used to rescheduling them (column 3).

In general, better performance is obtained with the local scheduler because the local scheduler generates code schedules that result in better utilization of hardware resources. In particular, the local scheduler resulted in a higher degree of concurrent utilization of both clusters, and a reduction in the number of dual-distributed instructions. In addition, with the local scheduler, instructions were issued more in order, with the effect that significantly fewer instruction-replay exceptions are required to free up an operand transfer buffer entry. One exception to this trend, however, is *ora*, for which the use of the local scheduler significantly increased the number of instruction replays, thus degrading performance. A more detailed analysis of the impact of the local scheduler is presented in [3].

Considering the ratios for the local scheduler indicates that its use results in a worst-case slowdown of 25%. To compensate for the increase in clock cycles that this slowdown represents, the dual-cluster processor would have to use a processor clock with a period 20% smaller ($= 100 -$

[2]The prediction is made at the point of insertion into the dispatch queue while the updating occurs after the branch is executed.

$100\frac{T_{dual}}{T_{single}} = 100 - 100\frac{N_{single}}{N_{dual}} = 100 - 100\frac{N_{single}}{1.25 \times N_{single}}$).

Recently Palacharla et al. have created delay models for the critical paths of dynamically scheduled superscalar processors [14] as a function of issue width. They report that in a 0.35μm process, the worst case delay increased from 1248ns for a four-issue processor to 1484ns for an eight-issue processor, an increase of 18%. Given that there is only an 18% difference between the cycle times of the four-issue and eight-issue processors in a 0.35μm process generation, reducing the cycle time through partitioning would not improve overall performance. However, for a 0.18μm process generation, Palacharla et al. found that the worst-case path would increase by 82% when moving from a four-issue processor to an eight-issue processor. The larger relative delays for wide-issue machines at 0.18μm feature sizes is due to wire delay increasing relative to gate delays as feature sizes are reduced. Thus, communication becomes relatively more expensive in comparison to computation. Given this larger cycle time difference between narrow and wide issue machines at smaller feature sizes, the net effect of partitioning in the multicluster architecture could result in a significant overall performance increase.

5 Conclusions

In this paper we have introduced the multicluster architecture, a decentralized, dynamically-scheduled architecture. In this architecture, the register files, dispatch queue, and functional units of the architecture are distributed across multiple clusters, and each cluster is assigned a subset of the architectural registers. The motivation for partitioning these resources is to reduce the size and complexity of components that are likely to be on the critical timing path of a centralized processor, and in the process, to reduce the cycle time of the decentralized processor.

The architecture provides a mechanism to allow the register file of one cluster to be accessed by instructions being executed on another cluster. This mechanism is based on distributing an instruction to more than one cluster for execution. The multiple distribution of instructions increases the number of clock cycles required to execute these instructions, and reduces the number of instructions that can be simultaneously in execution. However, these two negative effects are offset by the reduction in the cycle time of the processor clock that is provided by partitioning the register file and other hardware resources. Consequently, an application may require more clock cycles to execute on a multicluster processor than on a single-cluster processor, but its run time may be smaller.

There are two requirements for an application to perform well when executing on a multicluster processor. First, the instructions that are required to perform the task must be balanced across the clusters, with each cluster concurrently performing similar amounts of work. However, this requirement cannot be directly addressed by the compiler because the work done by a cluster is a function of the order in which instructions are issued, and the issue order is not deterministic. Thus, the compiler can only indirectly address the requirement by ensuring that the distribution of instructions to clusters is balanced. The second requirement is that the number of instructions distributed to more than one cluster must be minimized. This requirement seeks to counter the above noted two negative effects of multiple distribution. Since the distribution of instructions is based on the architectural registers named by the instructions, static instruction scheduling is the process by which the instructions are ordered and architectural registers are assigned to the operands and results of the instructions.

In this paper, we have described a novel algorithm we developed to implement this process. We have also described some of the results from simulations we performed on a number of dual-cluster processor configurations to evaluate the effectiveness of the algorithms. Using the processor cycle time analysis of Palacharla et al. for a 0.35μm process, the negative instructions-per-cycle effects of partitioning would slightly outweigh the advantage gained from a reduction in cycle time. However for smaller feature sizes, such as in their 0.18μm process model, a significant net performance improvement could be obtained. Thus we believe the multicluster architecture warrants further investigation.

6 Future Work

A promising area for further investigation are optimizations that require information derived from the source code of an application, which we did not consider owing to the limitations of the framework we used to implement the schedulers. For example, techniques such as superblock scheduling [15], and trace scheduling [5] might be used to increase the number of instructions that can be jointly scheduled, thus permitting a better estimation of the run-time distribution of the workload. Loop unrolling, which is a part of trace scheduling, could also be used to generate a code schedule in which multiple iterations of a loop were interleaved, with each iteration scheduled to use a separate cluster of a multicluster processor. To further increase the performance of loop unrolling, schemes could be devised to decrease the amount of interaction between the iterations of the loop, and thus, the number of inter-cluster data transfers. One such scheme is to duplicate the code that calculates addresses. A second scheme is to allocate key variables to global registers so that the variables can be accessed from within each cluster without an inter-cluster data transfer.

A second set of techniques might be used to exploit the hardware mechanism (see [3]) that was developed to permit the dynamic reassignment of the architectural registers to the clusters of a multicluster processor. In particular, the compiler could provide the hardware with hints to indicate when the reassignment could be made, and to directly specify the architectural-register-to-cluster assignment for each architectural register. This functionality would provide additional flexibility in separating a sequence of instructions into a number of partially-independent threads.

Acknowledgments

The research described in this paper has been partially funded by the Natural Sciences and Engineering Research Council of Canada and by Digital Equipment Corporation. We thank Brad Calder and Alan Eustace for helping us with the ATOM simulation infrastructure, Annie Warren and Jason Wold for logistical support while our simulations ran, and the other WRL-ites for putting up with these simulations. In addition, we thank the anonymous reviewers for their comments, and Subbarao Palacharla and Pritpal Ahuja for their early work on the architecture. Finally, we thank Digital Equipment Corporation for providing us with the Alpha AXP workstations.

References

[1] Linley Gwennap. Digital 21264 Sets New Standard. *Microprocessor Report*, 10(14), 1996.

[2] Kenneth C. Yeager. The MIPS R10000 Superscalar Microprocessor. *IEEE Micro*, 16(2):28–40, 1996.

[3] Keith I. Farkas. *Memory-system Design Considerations for Dynamically-scheduled Microprocessors.* PhD thesis, Department of Electrical and Computer Engineering, University of Toronto, Ontario, Canada, January 1997. (URL: http://www.eecg.toronto.edu/∼farkas/thesis_phd.html).

[4] James E. Smith. Decoupled Acess/Execute Computer Architecture. *In the Proceedings of the 9th International Symposium on Computer Architecture*, pages 112–119, 1982.

[5] P. Geoffrey Lowney, Stefan Freudenberger, Thomas Karzes, W.D. Lichtenstein, Robert P. Nix, John S. O'Donnell, and John C. Ruttenberg. The Multiflow Trace Scheduling Compiler. *Journal Of Supercomputing*, 7(1-2):51–142, May 1993.

[6] Gurindar S. Sohi, Scott E. Breach, and T. N. Vijaykumar. Multiscalar processors. *In the Proceedings of the 22st International Symposium on Computer Architecture*, pages 414–425, 1995.

[7] Dean M. Tullsen, Susan J. Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lo, and Rebecca L. Stamm. Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreaded Processor. *In the Proceedings of the 23rd International Symposium on Computer Architecture*, pages 191–202, May 1996.

[8] Basem A. Nayfeh, Lance Hammond, and Kunle Olukotun. Evaluation of Design Alternatives for a Multiprocessor Microprocessor. *In the Proceedings of the 23rd International Symposium on Computer Architecture*, pages 67–77, May 1996.

[9] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, Principles, Techniques and Tools*. Addison-Wesley Publishing Company, Reading Mass., 1986.

[10] Preston Briggs, Keith D. Cooper, and Linda Torczon. Improvements to graph coloring register allocation. *ACM Transactions on Programming Languages and Systems*, 16(3):428–455, May 1994.

[11] Amitabh Srivastava and Alan Eustace. Atom: A system for building customized program analysis tools. *In the Proceedings of the ACM SIGPLAN `94 Conference on Programming Languages*, March 1994.

[12] Keith I. Farkas and Norman P. Jouppi. Complexity/Performance Tradeoffs with Non-Blocking Loads. *In the Proceedings of the 21st International Symposium on Computer Architecture*, pages 211–222, 1994.

[13] Scott McFarling. Combining branch predictors. *DEC WRL Technical Note TN-36*, 1993.

[14] Subbarao Palacharla, Norman P. Jouppi, and James E. Smith. Complexity-Effective Superscalar Processors. *In the Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 206–218, 1997.

[15] Pohua P. Chang, Scott A. Mahlke, William Y. Chen, Nancy J. Warter, and Wen-mei W. Hwu. IMPACT: an Architectural Framework for Multiple-Instruction-Issue Processors. *In the Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 266–275, 1991.