Continuous Optimization

Brian Fahs Todd Rafacz
Sanjay J. Patel Steven S. Lumetta

Center for Reliable and High Performance Computing
University of Illinois at Urbana-Champaign

Abstract

This paper presents a hardware-based dynamic optimizer that continuously *optimizes an application's instruction stream. In continuous optimization, dataflow optimizations are performed using simple, table-based hardware placed in the rename stage of the processor pipeline. The continuous optimizer reduces computation tree height by performing constant propagation, reassociation, redundant load elimination, and store forwarding. To enhance the impact of the optimizations, the optimizer integrates values generated by the execution units back into the optimization process. Continuous optimization allows instructions with input values known at optimization time to be executed in the optimizer, leaving less work for the out-of-order portion of the pipeline. This feature can detect branch mispredictions earlier and thus reduce the misprediction penalty. In this paper, we present a detailed description of the hardware optimizer and evaluate it in the context of a contemporary microarchitecture running current workloads. Our analysis of SPECint, SPECfp, and mediabench workloads reveals that a large percentage of instructions can be executed early, many mispredicted branches can be recovered at the optimization stage, and most memory operations can have their addresses fully generated in the optimizer. These positive effects combine to provide performance speedups in the range of 0.98 to 1.28.*

1 Introduction

Over the last several years, dynamic optimization has become a popular topic in computer systems research because it offers opportunities beyond static compiler optimization through its ability to identify hot execution paths dynamically, thereby adapting to changing program behavior and input and providing performance improvements even for binaries compiled with aggressive static optimization. Many dynamic optimization systems [1, 3, 6, 8, 9, 10, 20, 23] share a common overall structure: they (1) select regions (functions, traces, hyperblocks, etc.) through some form of dynamic profiling, (2) apply optimizations to the selected regions, (3) cache the optimized versions, and (4) replace future dynamic occurrences of the original regions with the optimized versions. In this paper, we propose a dynamic optimization system, which we call *continuous optimization*, that does not require profiling of the instruction stream or caching of the optimized instructions. Instead, dataflow optimizations are applied to each fetched instruction using a table-based hardware optimizer.

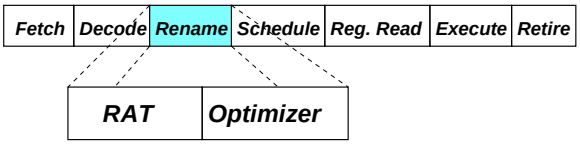

Figure 1. High-level view of optimizer

Figure 1 depicts the high-level organization of a continuous optimizer. It is integrated into the rename stage of a dynamically-scheduled processor, and uses a series of hardware tables described in Sections 2 and 3 to perform constant propagation, reassociation, redundant load elimination, and store forwarding. The optimizer also takes advantage of values already generated by the execution units to further optimize the instruction stream. This feedback path enables the optimizer to directly execute some fraction of instructions without having to send them into the out-of-order core. As a whole, the objectives of the optimizer are (1) to reduce computation tree height, and (2) to execute simple[1] instructions in the optimization stage. These two objectives are symbiotic and combine to provide the overall benefit of continuous optimization.

Previous work on hardware-based dynamic optimization [1, 23] performed optimization offline through an abstract optimizer that operated on discrete traces extracted from the instruction stream. Our architecture

[1] Simple instructions are those that require a single cycle to execute.

performs low-level compiler optimizations *continuously* on the whole instruction stream, but can easily be adapted for offline optimization frameworks.

In summary, several contributions are made in this paper:

- The notion of continuous optimization.
- A detailed description of a table-based hardware optimizer that implements low-level compiler optimizations.
- A quantitative analysis of the performance impact and sensitivity to various design choices for continuous optimization.

This paper is organized as follows. Section 2 provides the motivation and high-level details of continuous optimization. Section 3 presents the details of the hardware required for continuous optimization. Section 4 presents our experimental infrastructure. In Section 5, we provide our performance characterization, and in Section 6, we present our sensitivity studies. Section 7 presents related work. Section 8 provides a summary of the findings.

2 Continuous Optimization

Figure 2 contains a more detailed view of the continuous optimization hardware. Constant propagation (CP) and reassociation (RA) are implemented by augmenting the register alias table (RAT) with additional information. Redundant load elimination (RLE) and store forwarding (SF) are implemented using a separate, cache-like structure that is accessed after the RAT.

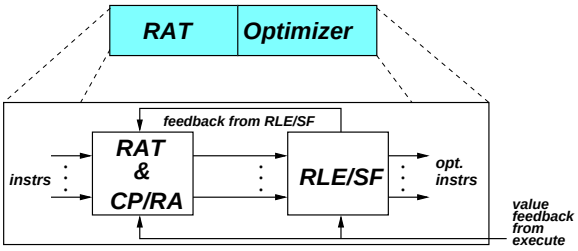

Figure 2. Architectural view of optimizer

In this architecture, the optimization process is applied to every dynamic instruction. Unoptimized instructions from the decode stage of the pipeline have their inputs and outputs renamed while simultaneously being transformed into more parallel instructions. The crux of the optimizer architecture is that a *symbolic* representation of the value of each architectural register is maintained within the RAT. The optimizer leverages this symbolic representation to expose opportunities for parallelism in the dataflow and to reduce the number of memory accesses.

As a further enhancement, values generated during execution are fed back to the optimizer in order to increase the availability of constant values for the optimizer. We call this process *value feedback*, and call the values fed back *known values* to differentiate them from the symbolic information available before an instruction has executed. In particular, these values are known at the time of their use in the optimizer, and can thus be treated as constants. The mechanism used for value feedback is similar to that already present for bypass, but differs in that the values returned to the optimizer are not necessary for correct behavior (symbolic values suffice for correctness), thus instruction optimization does not stall waiting for value feedback.

In this section, we describe the general operation of the optimizer and the concept of value feedback using a motivating example.

2.1 Optimizations

The optimizer considered in this paper performs four common dataflow optimizations: constant propagation, reassociation, redundant load elimination, and store forwarding. Each is briefly described below:

Constant propagation (CP) propagates known values from producers to consumers. This optimization reduces computation tree height by removing any dependences on known values, and also allows simple instructions with only known values as inputs to execute within the optimizer. If, for example, the instruction `addq r3, 4 -> r4` is to be optimized, and `r3` is known to have the value 3, the constant propagation logic performs the add and moves the value 7 into `r4`.

Reassociation (RA) flattens recursive expressions of the form `(reg << scale)` $\pm$ `offset`, reducing computation tree height and increasing parallelism by shifting dependences to earlier producers. For reassociation, all `scale` and `offset` values are constants extracted from instructions, while `reg` values are symbolic. The optimizer thus copies the symbolic `reg` value from producer to consumer and recalculates the consumer's `scale` and `offset` fields.

Redundant load elimination (RLE) combines two load operations accessing the same memory location into a single operation. The second load is converted to a move operation, which is then optimized away in a manner similar to [10, 15]. This optimization happens only when the physical destination of the first load still contains its value, i.e., it has not yet been assigned to another instruction.

Store forwarding (SF) converts a load operation that references a recently stored value into a move operation, which is then optimized away in the same manner as is used for redundant load elimination.

In our optimizer design, RLE/SF follows CP/RA.

Constant propagation and reassociation serve to flatten the `BaseReg` $\pm$ `offset` address specifications of memory instructions, enabling redundant load elimination and store forwarding to capture more instructions.

In addition to these optimizations, our system performs several other minor optimizations. The consumers of register move instructions are made dependent on the producer of the move instruction through reassociation. Additionally, simple strength reductions are performed when possible (multiplies by powers of two are converted into left shifts). Finally, if the direction of a branch indicates information about a register value (e.g., `beq` indicates that the register value is zero if the branch is taken), the register is assumed to be precisely that value. This optimization is safe because, in the event of a branch misprediction, the optimized instructions on the wrong path are discarded, and the optimization state is recovered.

2.2 Value feedback

Each value generated by an execution unit can be integrated into the optimization tables, thereby converting a symbolic value into a *known value*. The known value can then be propagated by the CP/RA logic into consumer instructions. For example, if the instruction `addq r3, 4 -> r4` executes and generates the result 15, and this particular version of `r4` is still architecturally live, the optimization tables record that `r4` contains the known value 15. Subsequent instructions that use `r4`, such as `beq r4, LOOP`, read the value 15 for `r4` and potentially execute in the optimizer.

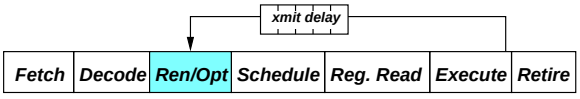

Figure 3. Value feedback from execution

Notice that this temporal notion of a known value depends on the pipeline length between the optimization stage and the execution stage, as indicated by Figure 3. That is, a symbolic value referencing an instruction's destination register is added to the optimization tables when the instruction is renamed. If this register is still being referenced in the optimization stage when the instruction executes (actually some small transmission latency after the instruction executes), the symbolic value is replaced with the actual value.

2.3 Performing other optimizations

We considered the implementation of several other dataflow optimizations in the context of a continuous optimizer, but decided not to include them for various reasons. We discuss these optimizations and reasons below.

Common subexpression elimination leverages the fact that unrelated operations (i.e., in terms of PC and data-flow dependence) occasionally compute identical values by forwarding the result of the first computation on to the second. General instruction reuse [24] performs this optimization by discovering instructions with identical operand/immediate and physical register dependence. However, general instruction reuse has limited effectiveness in that it does not simplify computations before comparison and, therefore, misses opportunities where common expressions are effectively but not computationally the same. Incorporating general instruction reuse would be beneficial, but its implementation differs significantly from continuous optimization. Redundant load elimination, one of our four optimizations, is a subset of common subexpression elimination, but we include it because it is a simple and natural extension to our store forwarding optimization.

Dead code elimination is a simple algorithm, but requires explicit knowledge of consumer and anti-dependence relationships. Since dead code elimination requires looking at the future execution stream, it requires additional instruction buffering, potentially requiring additional pipeline stages. Furthermore, speculation support is no longer trivial because the original future information following a branch misprediction will change. As a result, correcting a misprediction may require resurrecting instructions from before the branch that were previously identified as dead. With the optimizations included in our current design, all decisions are based on previously executed instructions, thus misspeculated instructions can be discarded with no ill effects. The current optimizations do substantially increase the fraction of dead instructions in the instruction stream, but few of these instructions are removed in our present evaluation. A mechanism to remove dead instructions from the pipeline, such as that proposed in [7], is likely to improve continuous optimization performance beyond what we show here.

2.4 Motivating example

To demonstrate the operation of continuous optimization, we use the simple code example in Figure 4. *Static Code* shows the static representation of a loop that sums together the elements of an array. For the purposes of this example, the loop counter is initialized to some value that is not statically computable. On each iteration of the loop, an array value is loaded and added to the sum. The loop counter is decremented, and the next array index is computed. The loop ends when the loop counter reaches zero. *Dynamic Data Flow* illustrates the producer-consumer relationships of the loop instructions as it executes within a processor that performs register renaming. On such a processor, each arc is bound to

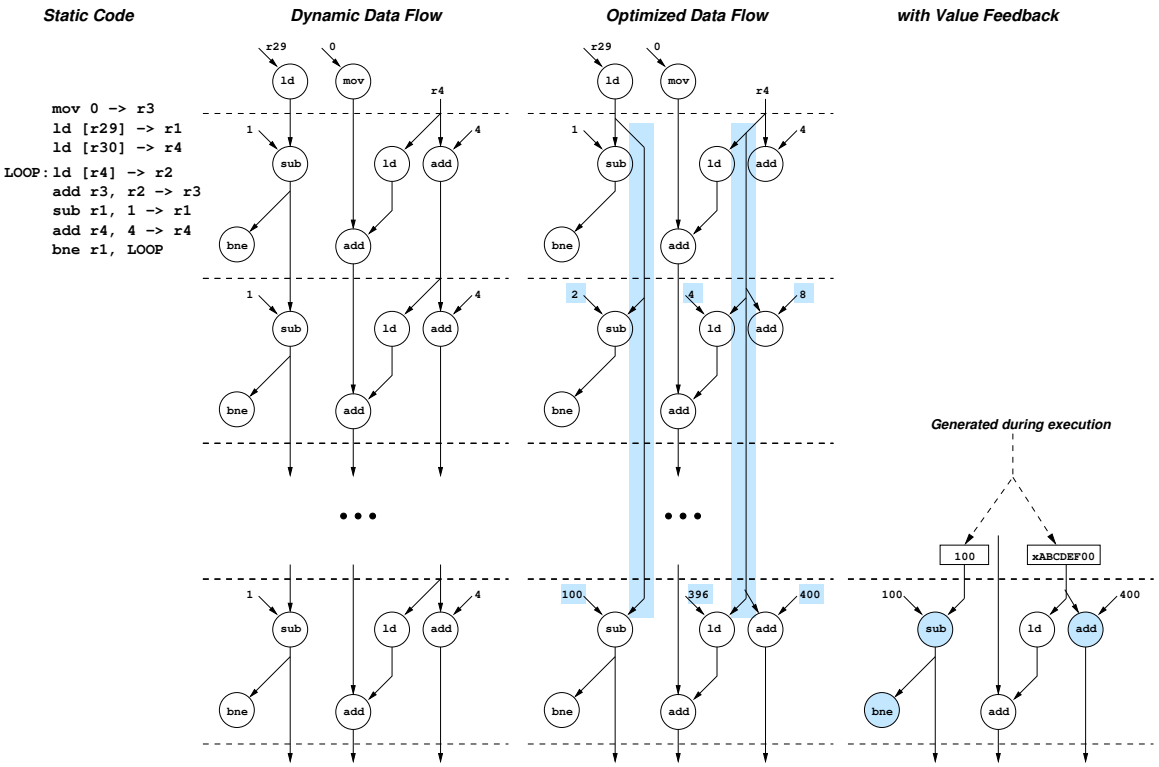

Figure 4. Motivating example

a physical register by the register renamer. Notice that each array index addition and array element load within each iteration is fed by a loop-carried dependence. Similarly, the branch at the end of each iteration is dependent on a chain of subtractions equal in length to the chain of array index additions.

The sequence of operations emitted by the continuous optimizer is represented by the column labeled *Optimized Data Flow*. The arcs that are modified by the optimization logic are shaded. Although the computation tree height of the accumulate chain is not reduced significantly, the chains of computation for the array index, load, and loop counter have been eliminated.

To make this process more concrete: consider what happens when the `SUB r1, 1 -> r1` instruction from the first iteration enters the rename/optimization stages of the pipeline. The RAT and CP/RA tables are accessed by the source architectural register number `r1`, and the previous mapping of `r1` is discovered (it was previously generated by the instruction `ld [r29] -> r1`). Say it is `p35`. Because the `SUB` instruction also writes to `r1`, the mapping for `r1` is updated with the symbolic value `p35 - 1`. When the `SUB r1, 1 -> r1` instruction from the second iteration is renamed, `r1` still maps to the symbolic value `p35 - 1`, and the optimizer replaces this mapping with `p35 - 2`. If the physical register destination chosen for this second SUB operation is p37, the instruction itself becomes `SUB p35, 2 -> p37`. This process is described in more detail in the next section.

By the time the 100^{th} iteration occurs, the instructions from the beginning of the loop are likely to have completed execution, in which case their results have been placed into the optimization tables (value feedback). The column labeled *with Value Feedback* shows the modifications to the 100^{th} iteration that result from incorporating the values of previously executed instructions into the optimizations. In particular, both the iteration counter load (`ld [r29] -> r1`) and the array base load (`ld [r30] -> r4`) are assumed to have completed execution and their results are integrated back into the optimization tables because both are still live. The shaded instructions are from iteration 100 and their outputs can be completely determined within the optimizer because all of their inputs are known values. The out-of-order portion of the pipeline does not need to execute these instructions. Also, the optimizer computes the memory address for the load instruction, enabling the load to proceed directly to the data cache read port.

This particular example does not make use of the RLE/SF portion of the optimizer, but the optimization process from this example applies directly to the redundant load and store-forwarding process.

2.5 Impact of continuous optimization

As with any new microarchitectural feature, both positive and negative aspects must be considered. We devote this subsection to discussing the potential impacts of continuous optimization.

2.5.1 Positives

From a qualitative point of view, continuous optimization can directly provide several benefits that, depending on the baseline processor microarchitecture, can improve performance and/or power.

Computation tree height reduction is provided by all four optimizations. It is beneficial because it results in more instruction-level parallelism (ILP). For example, as illustrated by the example in Section 2.4, some chains of dependent adds can be reduced to a single add.

Early execution, i.e., executing instructions in the optimization stage of the pipeline, has several positive benefits. First, it creates a synergistic effect: the resulting constant is propagated to consumers, which might also be allowed to execute early. Additionally, early execution relieves pressure on the out-of-order portion of the pipeline because instructions that are executed early only need to be retired, i.e., they do not need to pass through the scheduler, dispatch, register read, and execute stages. As we show in Section 5, roughly one in four instructions executes early. An important sub-case of this effect is early branch resolution. With continuous optimization, over 10% of mispredicted branches can be resolved at rename. The newest Pentium 4 [13] has a minimum branch misprediction penalty of over 30 cycles, the majority of which occur post-rename. Almost all post-rename cycles can be saved when a mispredicted branch is executed early.

Load reduction is provided by the redundant load and store forwarding optimizations. Both optimizations exploit provable short-term reuse of data to remove load instructions (i.e., loads to addresses which have recently been loaded or stored and whose value still exists in the physical register space). These optimizations can remove nearly 20% of load instructions, potentially reducing the power due to data accesses, since an optimizer table read is likely to consume less dynamic power than an L1 cache access.

2.5.2 Negatives

The positive aspects come at some cost, however. Here we list several negative aspects of continuous optimization.

Increased pipeline depth is one potential drawback of inserting an optimizer into a processor pipeline. As discussed in Section 3, the number of additional pipeline stages required for optimization is likely to be small, on the order of two to four stages. As the astute reader may notice, the addition of pipeline stages increases the misprediction penalty, and, conversely, early branch resolution reduces branch misprediction penalty for branches executed in the optimizer. These effects counteract one another, and the resulting performance impact depends on the number of branches that can be recovered early.

Design complexity potentially increases with continuous optimization. The increased rename complexity caused by the addition of the optimizer may be significant. As shown in the next section, fast, simple ALUs are required for each instruction that can pass through rename in a single cycle (e.g., four in a four-wide rename stage), plus additional forwarding and bypass logic to perform the symbolic optimizations. In addition, value feedback carries results from the execution stage back into rename. This feedback requires additional forwarding logic that does not exist in current processors. The continuous optimization tables require approximately 2K to 4K bytes of storage[2]: the CP/RA tables require one entry per integer architectural register, and each entry contains approximately 100–150 bits. This table requires as many read and write ports as required by the RAT to support the rename rate. The RLE/SF stage also requires a small cache, which we model as consisting of 128 entries, each requiring approximately 100-150 bits. This small cache requires a read and write port for each load capable of being optimized each cycle.

Power. The impact on overall power consumption is not obvious. The power implications of both the additional pipeline stages [12] and increased rename complexity is unclear. Certainly, without any simplification in the out-of-order portion of the pipeline, it can be argued that power requirements increase. However, the optimizations simplify and pre-compute many of the instructions, thus, there is an opportunity to reduce the complexity of the out-of-order pipeline, but this is a subject of future work.

3 Microarchitectural Details

In this section, we delve more deeply into the microarchitectural details of continuous optimization. The optimization process consists of two sequential steps. CP/RA is performed in the first step, which happens concurrently with register renaming. RLE/SF happens in the second step. The detailed microarchitecture is provided in Figure 5, which shows the logic slice needed to process one instruction in a multi-instruction rename bundle.

3.1 CP/RA

In order to perform constant propagation and reassociation, a small amount of additional informa-

[2]We have not yet analyzed any tradeoffs between storage and performance. We believe that significant storage reduction is possible.

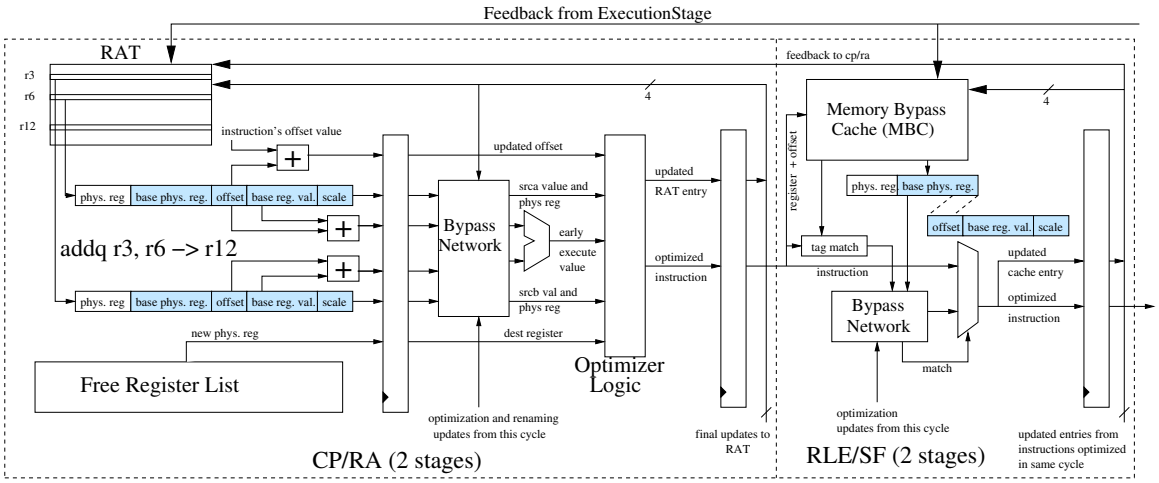

Figure 5. A microarchitecture diagram of the optimizer showing the logic slice necessary to optimize a single instruction

tion must be maintained per architectural register. Specifically, we maintain information about whether the register contains an expression of the form `(reg << scale) ± offset`. Here, the register specifier `reg` indicates a physical register, `scale` indicates a two-bit left shift quantity, and `offset` is a 64-bit immediate field. When an instruction enters the rename stage of the pipeline, its input source operands access the register alias table (RAT) and read the renamed register mapping as well as a symbolic value for each source, if one exists. The optimizer then processes the symbolic source information and either (1) immediately determines the output of the instruction, (2) derives a new symbolic value for the destination of the form `(reg << scale) ± offset`, or (3) determines that the symbolic form of the output is too complex, in which case its value must be generated by the execution core. The result is stored in the RAT for future instructions[3].

From an implementation point of view, constant propagation and reassociation are treated as equivalent transformations by the optimizer hardware. To denote that the output of an instruction is a constant value, the register field of the symbolic form `(reg << scale) ± offset` is set to the hardwired zero register, and a full 64-bit data value is stored in a "base register value" field.

Processing a symbolic value requires that the optimizer shift the base physical register value by the scale, if both are present. The scaled constant and the offset are then passed as inputs to an ALU, which fully executes the instruction if all values are known. Otherwise, depending on the opcode, the optimizer may produce a new offset value and/or physical register inputs for the instruction, or it may produce nothing if the instruction's result cannot be encoded in the symbolic representation. The destination register's RAT entry is modified accordingly.

Careful consideration reveals that extending the optimizer to process multiple instructions per cycle can be a little problematic. In particular, the destination updates produced by optimizing one instruction may be required as input for another instruction in the same rename bundle. For a four-wide renamer, the implication is that four serial additions are required. In our evaluation, we allow only a single level of addition to occur for any fetch packet. Instructions that require more than one level of addition are not optimized. For example, consider the following instruction sequence, assuming that the RAT entry for r1 indicates that it is symbolically `r0 + 1`.

```
add  r1, 1 -> r2
add  r2, 1 -> r3
add  r3, 1 -> r4
add  r4, 1 -> r5
```

The optimizer can in theory optimize this sequence into four parallel instructions, but only through multiple serial additions. Instead, we limit the number of additions that can occur to one, and only the first instruction is reassociated, as shown below. The performance implications of this choice are discussed in Section 6.2.

```
add  r0, 2 -> r2
add  r2, 1 -> r3
add  r3, 1 -> r4
add  r4, 1 -> r5
```

[3]Again, because the optimizer only contains simple ALUs, only one-cycle instructions execute directly in the optimizer.

A second subtlety arises regarding the lifetime of physical registers. Many current physical register allocation/deallocation schemes, such as those used in the MIPS R10000 and Alpha 21264, allow physical registers to be reused after the retirement of the next instruction that overwrites the architectural destination. Our optimizations can extend the lifetime of a physical register value beyond this point, in which case such deallocation schemes no longer work. We instead rely upon an algorithm based on reference counters, such as the scheme proposed by [15].

Finally, it is important to notice that a single physical register may be referenced several times in the RAT (i.e., once at its destination's architectural register and possibly many times as a base physical register). This multiplicity complicates the mechanism of value feedback, in which the value of an executing instruction is integrated into the optimization table. We propose a method for providing value information to multiple consumers within the RAT in Section 3.3.

3.2 RLE/SF

Redundant load elimination and store forwarding only perform transformations on load instructions. A small cache that maintains information about previous load and store instructions is used to match load instructions passing through the pipeline with previous memory instructions that accessed the same memory location. Once a match is found, the load instruction is converted into a move operation that references the previous memory operation. All instructions dependent on the load then use the destination (or source, if a store) of the previous memory operation as their source.

A complication arises because, at rename, memory addresses are generally not known. The RA/CP step increases the number of memory instructions whose memory addresses can be computed at rename to nearly 70%. When a load is encountered with a known address, it is looked up in a small table called the Memory Bypass Cache (MBC). A hit provides the symbolic representation of the data for the load, no subsequent cache access is required. If the load address is unknown, no optimization is performed. If a store instruction passes through the pipeline with an unknown address one of two things must be done: (1) flush the Memory Bypass Cache because of consistency issues, or (2) proceed speculatively and recover if the store happens to collide with an existing entry. For all evaluations presented in this paper, our continuous optimizer and pipeline proceeds speculatively. However, we have evaluated both scenarios and have found little difference in the overall performance.

The Memory Bypass Cache (MBC) component of the diagram is a small and constrained cache. Excluding the access information, the cache line data is precisely the same data provided by the RAT. To simplify the table, it is assumed that entries are all 8-byte aligned. The access information for tag matching not only needs to match the standard address tag but also match the offset from the 8-byte alignment and the size of the memory access. If a load address matches an entry in the table, the cache line data (RAT entry) is forwarded to all intermediate references (instructions in the current and previous pipeline stages), written back into the CP/RA table updating the load destination information, and used for converting the load instruction into the expression provided by the cache line. If the load address does not hit in the MBC, the MBC entry for the load address is updated to reference the physical register destination of the load in case that another load to the same address follows after this instruction. For store instructions, the data source information provided by the CP/RA table is stored in the MBC. As with multiple sequential additions for CP/RA, we do not allow any dependences across instructions within a rename packet to be satisfied with RLE/SF. In our experimentation, we use an MBC consisting of 128 entries.

3.3 Value Feedback

With Value Feedback, the results of executed instructions are propagated back to the optimization tables to further enhance the ability of the optimizer to pre-determine instruction outputs. This is problematic for the optimizer architecture presented in this section because a single physical register may be referenced multiple times within the optimization tables (RAT and MBC). In order to update multiple table locations simultaneously with the same physical register value, either a content-addressable structure is necessary or a level of indirection is required. When a physical register value is produced, each table entry can perform a content-addressable match with the "base physical register" field. If there is a match, the "base physical register" field is set to the zero register and the "base physical register value" is updated with the corresponding value. With the indirection approach, the "base physical register" field can be examined in a separate value table, but this adds extra latency to the optimizer, potentially complicating the intra-optimizer bypass network.

3.4 Continuous online versus discrete offline optimization

Although this hardware has been described in the context of continuous optimization, the actual hardware structures can easily be adapted for offline hardware-based optimization schemes such as rePLay [23], PARROT [1], or trace-cache-based schemes [10, 14]. The basic structure would remain similar. The major fun-

damental difference between the online and offline optimizations is that the optimization table entries would be invalidated at the start of each trace (or frame). That is, the optimizations are discrete per region as opposed to continuous across the execution stream. Furthermore, real-time value feedback for discrete optimization is more difficult. On the other hand, multipass and complex optimizations such as dead code removal and common subexpression elimination are more easily considered in the context of offline optimization.

4 Experimental Setup

4.1 Experimental Workload

For all experimental evaluation, we use the SPEC2000 integer, SPEC2000 floating point, and mediabench benchmarks. The specific benchmarks that were included and the simulated instruction counts for each benchmark are provided in Table 1. We included all benchmarks that our infrastructure would accommodate. For most of the SPEC integer benchmarks, the input sets were modified to allow simulation through completion of the program. The benchmarks written in C were compiled with the Compaq Alpha C compiler, Compaq C V5.9, with optimization level 4. The C++ and Fortran77 benchmarks were compiled at the highest optimization level with g++ and g77, respectively.

Type of App.	Name		Total Insts.
SPECint	bzip2	(bzp)	293M
	crafty	(cra)	625M
	eon	(eon)	132M
	gap	(gap)	474M
	gcc	(gcc)	284M
	mcf	(mcf)	410M
	perlbmk	(prl)	1000M
	twolf	(twf)	596M
	vortex	(vor)	272M
	vpr	(vpr)	1000M
SPECfp	ammp	(amp)	500M
	applu	(app)	382M
	art	(art)	1000M
	equake	(eqk)	1000M
	mesa	(msa)	1000M
	mgrid	(mgd)	1000M
mediabench	g721 decode	(g721d)	662M
	g721 encode	(g721e)	358M
	mpeg2 decode	(mpg2d)	220M
	mpeg2 encode	(mpg2e)	1000M
	untoast	(untst)	96M
	toast	(tst)	287M

Table 1. Experimental Workload

Fetch/Decode/Rename	4 insts/cycle
Retire	6 insts/cycle
BrPred	18-bit gshare, 1K-entry BTB
Pipeline	20 cycles (min) for BR res (if not executed early)
Scheduler	four 8-entry schedulers (int, complex int, fp, mem)
Inst Window	max. 160 in-flight insts
ExeUnits	4 Simple IALUs, 1 Complex IALU, 2 FPALUs, 2 Agen
L1 I Cache	64KB, 4-way assoc., 64B line size, 1 cycle
L1 D Cache	32KB, 2-way assoc., 32B line size, 2 ports, 2 cycles
L2 Unified Cache	1MB, 2-way assoc., 128B line size, 10 cycles
Memory	100 cycle latency
Optimizer	2 stages, Memory Bypass Cache of 128 entries, 4 rd/4wr ports

Table 2. Simulated Machine Configuration

4.2 Performance model

The SimpleScalar 3.0 tool set provides the framework on which our machine model is built. Our custom timing model resembles the pipeline of the Pentium 4 described in [13]. The specifics of the machine model are provided in Table 2. These default configurations should be assumed for all experiments unless stated otherwise.

In addition to the default processor configurations, there are also several default settings pertaining to the optimizer. Our baseline machine configuration (without continuous optimization) has two fewer pipeline stages in rename. Therefore, continuous optimization has an additional two cycle branch misprediction penalty for mispredicted branches that are not resolved in the optimizer. For those mispredicted branches that are resolved early, recovery happens after the extended rename stage. We assume that execution results being fed back to the optimizer incur a one cycle transmission delay. Our default optimizer configuration only evaluates a single level of addition dependence in a cycle. Therefore, if one addition feeds another addition within a rename bundle, the dependent instruction will not be optimized. Similarly, if the result of one load is used for the address of another load within a rename bundle, the dependent instruction will not be able to be optimized. The sensitivity of these default values is evaluated in Section 6.

For all of our optimizations, correctness is verified through strict expression and value checking to ensure

that faulty optimizations are not performed.

5 Performance Characterization

In this section of the paper, we evaluate several performance aspects of continuous optimization, both on our baseline processor configuration and on variations of the baseline.

5.1 Speedup over the Baseline

Figure 6 demonstrates the performance of continuous optimization for all of the SPECint, SPECfp, and mediabench benchmarks. The horizontal axis specifies the benchmark, and the vertical axis shows speedup over the default processor configuration without optimization. The average performance improvement is shown as the rightmost bar in each graph. Mediabench shows the largest overall performance improvement. The most important observation that can be made from these graphs is that, despite the additional pipeline stages, almost all benchmarks are able to demonstrate a performance improvement. Speedups range from 0.98 to 1.28. Note: the benchmark amp exhibited a speedup of 1.00.

Several of the benchmarks demonstrate significant improvements. We devote Section 5.2 to analyzing the performance improvements observed for mcf (mcf) and untoast (untst) because these benchmarks demonstrated the largest speedups.

Benchmark	exec. early	recov. mispred. brs.
SPECint	20.0%	10.5%
SPECfp	28.6%	17.5%
mediabench	33.5%	13.5%
avg	**26.0%**	**12.2%**
Benchmark	ld/st addr. gen.	lds removed
SPECint	56.2%	5.5%
SPECfp	71.2%	21.7%
mediabench	84%	47.2%
avg	**65.3%**	**17.4%**

Table 3. Effects of continuous optimization

In addition to reducing the number of cycles required for execution, continuous optimization can improve other characteristics. Table 3 presents some statistics. *Exec. early* is the percentage of the instruction stream that the optimizer was able to execute. *Recov. mispred. brs.* is the percentage of mispredicted branches that were able to be resolved and thus recovered in the optimizer. *Ld/st addr. gen.* is the percentage of all load and store instructions that were able to have their addresses generated in the optimizer. *Lds removed* is the average percentage of load instructions that were able to be converted into move operations by forwarding a value from a previous load or store operation. Across all benchmarks, 26% of instructions were executed in the optimizer, 12.2% of mispredicted branches were resolved faster, 65.3% of memory access instructions could have their address generated in the optimizer, and 17.4% of load instructions were executed in the optimizer.

5.2 Individual benchmark performance

Among the SPECint benchmarks, mcf provides an improvement two or three times as large as its peers. Because of this anomaly, we tracked the performance improvements to the source code level. One of the most significant performance improvements came from the **sort_basket** function in the mcf benchmark. The **sort_basket** function is an implementation of the well-known *quicksort* algorithm. To sort using the *quicksort* algorithm, a pivot point is chosen, and the array is partitioned around the pivot by sorting every other value only with respect to the pivot. The algorithm then recurses on the two unsorted sub-arrays created by the partitioning process. Continuous optimization produces performance from two sources for the *quicksort* algorithm. First, many instructions for the internal iterative loop of this algorithm can be executed in the optimizer. Dynamically, this early execution allows the instructions to exit the machine at a higher average rate. Second, the redundant memory accesses performed in *quicksort* create opportunities for redundant load elimination and store forwarding. Since the *quicksort* algorithm touches every element of the array at each level of recursion, the *quicksort* algorithm effectively fills up the MBC with array elements. Once the array being passed to *quicksort* is small enough that it does not thrash the MBC, all array accesses are eliminated, and the simple instructions dependent on these load operations are executed in the optimizer.

Continuous optimization also produces a speedup for untoast (untst) that is significantly higher than any other mediabench benchmark. The function **Short_term_synthesis_filtering** is one of the most significant performance contributors. This function is an iterative procedure that uses two 8-entry arrays. The loop iterations vary from 13 to 120 with each iteration performing some computation on every entry of the two arrays. Because the arrays are small enough to fit in the MBC, after the first iteration, all of the array accesses for this function are eliminated, and many of the simple instructions involved in the computation are performed in the optimizer.

5.3 Performance on other Machine Models

For the evaluations presented in this paper, we chose a baseline processor model that was relatively balanced;

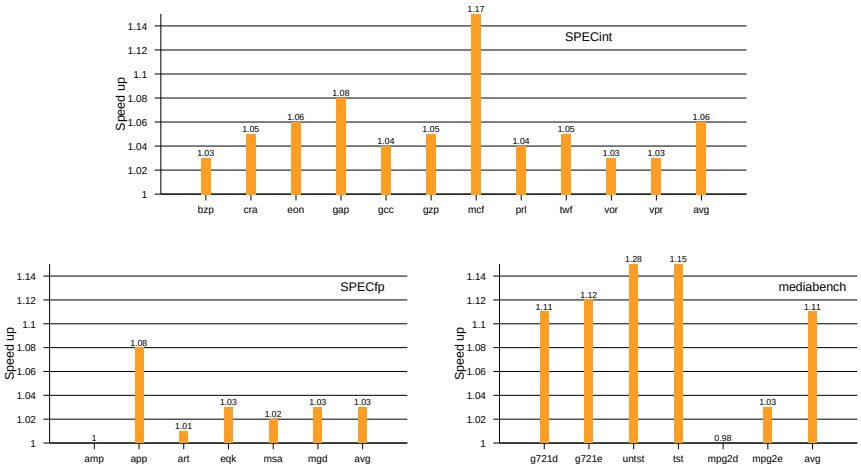

Figure 6. Speedup of continuous optimization over baseline

that is, in our estimation it was neither fetch-bound nor execution-bound. However, continuous optimization changes the overall balance of a machine, potentially changing an execution-bound machine into a balanced or fetch-bound machine effectively executing some fraction of the instruction stream at optimization time. Because continuous optimization allows some mispredicted branches to be resolved earlier, it also improves the fetch throughput. In this section, we evaluate the impact on processor throughput by observing the effect of continuous optimization on different processor configurations.

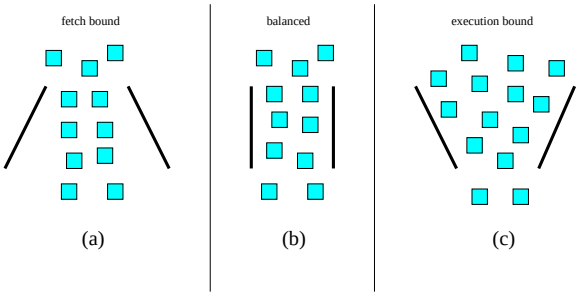

Figure 7. Processor throughput diagram

Figure 7 presents a high level view of the different machine configurations we evaluate. Figure 7(a) depicts a machine with ample execution resources, in which the fetch/decode/rename portion of the pipeline is limiting the overall performance. Figure 7(b) depicts a balanced machine, much like our default processor configuration, which is equally restrictive at fetch and execute. Figure 7(c) depicts a machine for which performance is restricted by the execution throughput.

Figure 8 demonstrates the performance of the three processor configurations previously described relative to the default processor configuration from Section 4.2. On

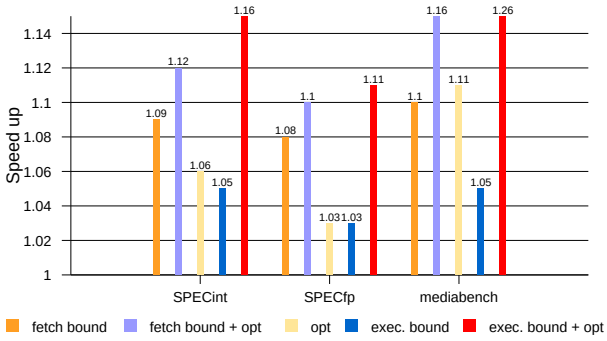

Figure 8. Performance relative to various machine configurations

the horizontal axis, there are five bars for each benchmark suite. *Fetch bound* is the performance of the default processor configuration where the processor is made fetch-bound by doubling the number of scheduler entries from four 8-entry schedulers to four 16-entry schedulers. *Fetch bound + opt* is the performance of *fetch bound* with continuous optimization. *Opt* is the performance of the baseline configuration from Section 4.2 with continuous optimization. *Exec. bound* is the performance of the default configuration where the processor is made execution-bound by changing the fetch/decode/rename from 4-wide to 8-wide. *Exec. bound + opt* is the performance of this configuration with continuous optimization.

The fetch-bound configuration has a significant speedup over the baseline for all benchmark suites. When continuous optimization is incorporated into the fetch-bound processor, a speedup occurs, but, the rela-

tive improvement over the fetch-bound configuration is much smaller than when continuous optimization is applied to the base configuration. Because the architecture is primarily fetch-bound, the benefits of executing instructions in the optimization stage of the processor are not as considerable. Under this configuration, resolving mispredicted branches in the rename stage of the pipeline is more important and thus provides the majority of the performance improvement demonstrated.

On the contrary, the speedup for continuous optimization in the execution-bound processor model is much more significant. In all benchmark suites, continuous optimization for the execution-bound processor offers average improvements of three to five times the improvement obtained by only widening the fetch bandwidth. This performance improvement demonstrates the ability of continuous optimization to increase the execution bandwidth of the processor, possibly without the complexity required to provide additional execution resources. Continuous optimization can thus be coupled with known techniques for scaling fetch bandwidth to produce a balanced processor with improved performance.

One interesting trend in the graph is that, for all of the benchmarks suites, continuous optimization with the default, balanced, processor model achieves speedups greater than or equal to the speedups achieved by doubling the fetch width of the processor. For the mediabench suite, continuous optimization also produces speedups greater than the speedups achieved from doubling the number of scheduler entries.

6 Performance Sensitivities

In this section of the paper, we evaluate the sensitivity of the continuous optimizer to various implementation considerations such as latencies and algorithmic tradeoffs.

6.1 Optimization versus Value Feedback

First we evaluate the contribution from value feedback versus optimization. Figure 9 presents the speedup observed for each overall benchmark suite when only value feedback is enabled versus when both value feedback and optimization are enabled. Value feedback alone, in some sense, can be considered a form of eager bypassing from the execution units back to the rename stage. It is clear from the experimental data that feedback alone offers little in terms of performance. Optimization enhances the effectiveness of feedback by projecting the usefulness of old values further into the future.

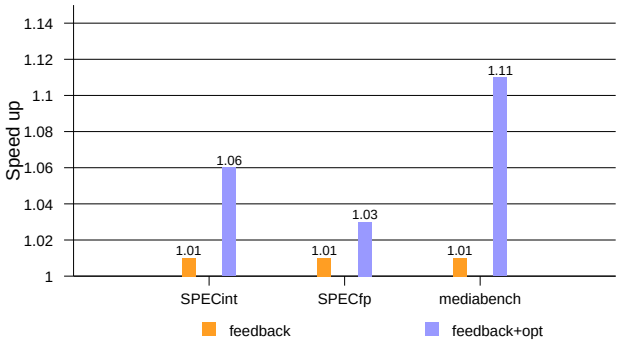

Figure 9. Continuous optimization vs. value feedback

6.2 Dependence depth

As discussed previously, since the optimizer must process multiple instructions in parallel (i.e., four in a four-wide machine), it may not be possible to optimize all instructions to the fullest degree when they are within the same rename bundle. In our default configuration, the optimizer handles only the first instruction in a chain of dependent additions. Also, for chained memory accesses, only the first instruction in the chain can query the MBC. In this section, we evaluate the missed opportunities that result from our conservative decisions. Specifically, we evaluate three additional scenarios: (1) up to one level of chained additions, (2) up to three levels of chained additions, and (3) up to three levels of chained additions and up to one chained memory operation.

Figure 10 shows four different bars corresponding to our default continuous optimization configuration and the three new scenarios described previously. For SPECint and SPECfp, there is very little performance to be gained from processing dependent instructions in parallel. However, mediabench shows a significant dependence on the ability to process multiple dependent instructions in parallel. Specifically, the speedups between handling dependent chains of additions up to length 3 increases the speedup on mediabench from an average of 1.11 to 1.25. There appears to be no additional benefit across all the benchmark sets in allowing chained memory operations. It should be noted that better compiler scheduling of rename bundles can potentially provide speedups similar to the “depth 3” case without additional hardware complexity.

6.3 Optimizer latency

Up until now, the additional pipeline stages required by the optimizer has been assumed to be two stages

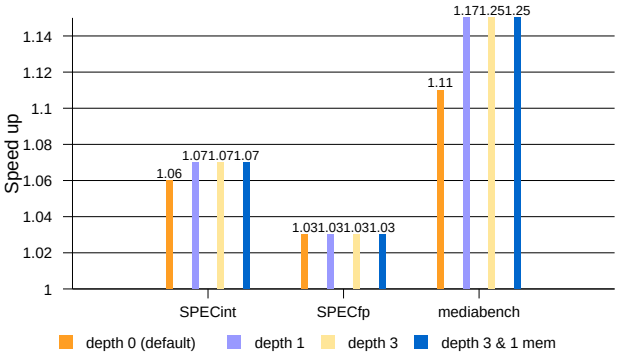

Figure 10. Importance of processing dependent instructions in parallel

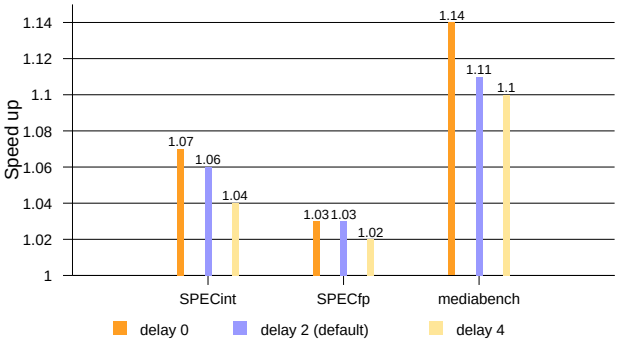

Figure 11. Optimizer latency sensitivity

out of an overall pipeline depth of 20 stages (for mispredicted branches). For this study, we evaluate the performance sensitivity of continuous optimization on optimization latency. Since optimization occurs online, this latency elongates the branch recovery critical loop, adding a negative component on performance. As shown in Figure 11, the achievable performance of continuous optimization can vary based on the number of additional pipeline stages, but it does not vary wildly and it is important to note that even at four additional pipeline stages (i.e., $\frac{1}{5}$ of total branch recovery loop), average speedup is still noteworthy, ranging from 1.04 to 1.10.

6.4 Value feedback latency

Implementation constraints, wire delays, and floorplan issues may complicate the delivery of value information from the execution units back to the optimization tables. Value transmission might take multiple cycles as a result. This delay could potentially have an impact on the performance of value feedback. If the delay is too long, the physical register might no longer be

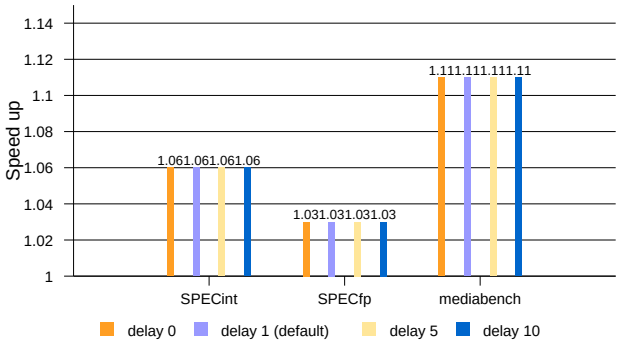

Figure 12. Performance sensitivity to value feedback transmission delay

referenced by new instructions, and therefore of no use in optimization. Our default configuration assumes that once an instruction executes, it will be available for optimization one cycle later. For this study, we also evaluate value transmission delays of zero, five, and ten cycles. Figure 12 demonstrates that there is no change in the overall performance resulting from additional delay. The key insight here is that either a physical register will be referenced by the optimizer for a long period of time or will not be referenced at all. For example, recall the motivating example from Section 2.4: the initial loop counter load and array base address were both loaded from memory at the beginning of the loop. The optimization process extended the live range of these values to include all iterations. The transmission delay of the values from execution therefore is of minimal impact.

7 Related Work

This paper contains a detailed description of a hardware optimizer that implements several low-level compiler optimizations and is general enough to be used for substantial online and offline dataflow optimization. Several previous works provide restricted variations of this architecture or concepts orthogonal to it. We describe these works below.

7.1 Early Address Resolution

Zero-cycle Loads [2] and Early Address Resolution [5, 11] aim to reduce the load-to-use latency by precomputing load addresses early in the pipeline. Continuous optimization is a generalization of these works. It executes as many simple instructions as possible in the early stages of the pipeline and reduces the computation tree height for the rest of the instruction stream that must proceed through the full pipeline. It has all of the advantages of these previous works and is also capable of optimizing the dependents of load instructions,

correcting branch mispredictions earlier, and also significantly reducing the number of instructions that must pass through the out-of-order portion of the pipeline.

7.2 Physical Register Reuse

Physical Register Reuse [15] exploits value redundancy in applications. Instructions that are deemed to produce an identical value to a previous instruction are mapped to the original producer's physical register instead of a new physical register. This optimization collapses dependency chains by redirecting dependent instructions to the original value producer rather than the duplicate value producer. The original work evaluated the impact of performing this optimization on direct moves and also explored the idea of implementing more advanced versions, both safe and speculative. General Instruction Reuse and Reverse Integration [24] extend this work by implementing advanced techniques for discovering value redundancy. Continuous optimization naturally detects and optimizes a significant portion of value redundancy (exceptions were discussed in Section 2.3), and it aggressively reduces computation tree height when value redundancy is not present. The primary focus of continuous optimization is not to detect value redundancy but rather pre-execute and reduce computation tree height.

7.3 Load and Store Reuse

Load and Store Reuse [22] is an alternative approach for performing redundant load elimination and store forwarding in hardware. Unlike continuous optimization, Load and Store Reuse performs optimization late in the pipeline and is unable to optimize the dependents of load instructions. Additionally, the RLE/SF stage proposed here allows data to be represented symbolically, which means data does not have to be ready prior to optimization. Since the RLE/SF stage stores its contents in the MBC and the MDRD stage of Load and Store Reuse relies on the contents of the memory location existing in a physical register indicates that the two approaches work under different circumstances implying they are at least partially orthogonal.

7.4 Similarities to Other Works

There are several previous works which share a similar framework to continuous optimization. We discuss these works here.

Flea-Flicker [4] and continuous optimization both pre-execute instructions early in the pipeline. Flea-Flicker is primarily designed to absorb L1 cache miss latencies and overlap L2 cache miss latencies that a compiler could not anticipate. It does this by speculatively pre-executing instructions without stalling in an Advance pipeline and then finalizing execution in a Backup pipeline which does stall. Continuous optimization not only pre-executes instructions but also optimizes those that it cannot execute. It is likely that incorporating continuous optimization into the Flea-Flicker pipeline would improve performance by reducing the computation tree height for the Backup pipeline.

Like continuous optimization, Physical Register Inlining [18] incorporates physical register values into the RAT. It does so only when the values require no more bits than the physical register tags.

Offline hardware optimization schemes such as rePLay [23], PARROT [1], and other trace optimization [10, 14] bear similarity to the online continuous technique, as we discussed in Section 3.4. The hardware for continuous optimization, although described here as a portion of the execution pipeline, can very easily be adapted to be a forward optimization pass in an offline optimizer.

In addition to these works, there are several other complimentary works. Scheduling optimizations such as those proposed in [16] can also be layered onto the continuous optimization performed by our hardware. In [25], the authors propose a technique for reducing power consumption by avoiding updates to the architectural register file when a value is detected to be short lived. Due to the extended physical register lifetimes, modifications to the proposed scheme are necessary. Early register deallocation [21, 19] or techniques to detect and remove dead instructions [7] will be enhanced by the ability of the optimizer to increase the fraction of dead code.

8 Conclusions

In this paper, we present and evaluate the concept of continuous optimization. Our table-based continuous optimizer can be integrated into the rename stage of a dynamically-scheduled processor. It performs simple dataflow optimizations by representing the results of instructions symbolically. In particular, the optimizer is able to reassociate and propagate constant values and perform store forwarding and redundant load elimination. We also enhance the optimizer with the ability to integrate known values (ie., values generated during execution) back into the optimization process. The optimizer requires a modest hardware budget requiring approximately 2KB–4KB of additional multiported storage in the rename stage along with 4 simple ALUs.

Through our evaluations, we found that continuous optimization can produce speedups ranging from 0.98 to 1.28 on a deeply pipelined processor similar to the Pentium 4. If the processor model has increased fetch bandwidth (say, through a trace cache), higher speedups are possible. We find that the optimizer is able to ex-

ecute many instructions, resolve branch mispredictions, and determine load addresses and values at rename time, potentially lessening the dynamic burden placed on the execution core.

9 Acknowledgements

We thank the other members of the Advanced Computing Systems group. This material is based upon work supported by the National Science Foundation under Grant Nos. 0092740 and 9984492 with very gracious support from Intel Corporation and Sun Microsystems.

References

[1] Y. Almog, R. Rosner, N. Schwartz, and A. Schmorak. Specialized dynamic optimizations for high-performance energy-efficient microarchitecture. In *Proceedings of the 2nd International Symposium on Code Generation and Optimization*, pages 137–148, 2004.

[2] T. M. Austin and G. S. Sohi. Zero-cycle loads: Microarchitecture support for reducing load latency. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 82–92, 1995.

[3] V. Bala, E. Duesterwald, and S. Banerjia. Transparent dynamic optimization: The design and implementation of Dynamo. Technical Report HPL-1999-78, Hewlett-Packard Laboratories, June 1999.

[4] R. D. Barnes, E. M. Nystrom, J. W. Sias, S. J. Patel, N. Navarro, and W. mei W. Hwu. Beating in-order stalls with "flea-flicker" two-pass pipelining. In *Proceedings of the 36th Annual International Symposium on Microarchitecture*, pages 387–398, 2003.

[5] M. Bekerman, A. Yoaz, F. Gabbay, S. Jourdan, M. Kalaev, and R. Ronen. Early load address resolution via register tracking. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 306–315, 2000.

[6] D. Bruening, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimization. In *Proceedings of the 1st International Symposium on Code Generation and Optimization*, pages 265–275, 2003.

[7] J. A. Butts and G. S. Sohi. Dynamic dead-instruction dectection and elimination. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 199–210, 2002.

[8] H. Chen, W.-C. Hsu, J. Lu, P.-C. Yew, and D.-Y. Chen. Dynamic trace selection using performance monitoring hardware sampling. In *Proceedings of the 1st International Symposium on Code Generation and Optimization*, pages 79–90, 2003.

[9] W. Chen, S. Lerner, R. Chaiken, and D. Gilles. Mojo: A Dynamic Optimization System. In *3rd Workshop on Feedback-Directed and Dynamic Optimization*, 2000.

[10] D. H. Friendly, S. J. Patel, and Y. N. Patt. Putting the fill unit to work: Dynamic optimizations for trace cache microprocessors. In *Proceedings of the 31st Annual International Symposium on Microarchitecture*, pages 173–181, 1998.

[11] S. Gochman, R. Ronen, I. Anati, A. Berkovits, T. Kurts, A. Naveh, A. Saeed, Z. Sperber, and R. C. Valentine. The Intel Pentium M processor: Microarchitecture and performance. *Intel Technology Journal*, 07(02):21–36, May 2003.

[12] A. Hartstein and T. R. Puzak. Optimum power/performance pipeline depth. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 117–128, 2003.

[13] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel. The microarchitecture of the Pentium 4 processor. *Intel Technology Journal*, Q(1), 2001.

[14] Q. Jacobson and J. E. Smith. Instruction pre-processing in trace processors. In *Proceedings of the 5th International Symposium on High Performance Computer Architecture*, 1999.

[15] S. Jourdan, R. Ronen, and M. Bekerman. A novel renaming scheme to exploit value temporal locality through physical register reuse and unification. In *Proceedings of the 31st Annual International Symposium on Microarchitecture*, pages 216–225, 1998.

[16] I. Kim and M. H. Lipasti. Implementing optimizations at decode time. In *Proceedings of the 29th annual international symposium on Computer architecture*, pages 221–232, 2002.

[17] H.-H. S. Lee, M. Smelyanskiy, G. S. Tyson, and C. J. Newburn. Stack value file: Custom microarchitecture for the stack. In *Proceedings of the 7th International Symposium on High Performance Computer Architecture*, pages 5–14, 2001.

[18] M. H. Lipasti, B. R. Mestan, and E. Gunadi. Physical register inlining. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, 2004.

[19] J. Martinez, J. Renau, M. Huang, M. Prvulovich, and J. Torrellas. Cherry: Checkpointed early resource recycling in out-of-order microprocessors. In *Proceedings of the 35th Annual International Symposium on Microarchitecture*, 2002.

[20] M. C. Merten, A. R. Trick, E. M. Nystrom, R. D. Barnes, and W. W. Hwu. A hardware mechanism for dynamic extraction and relayout of program hot spots. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, 2000.

[21] M. Moudgill, K. Pingali, and S. Vassiliadis. Register renaming and dynamic speculation: An alternative approach. In *Proceedings of the 26th Annual International Symposium on Microarchitecture*, pages 202–213, 1993.

[22] S. Önder and R. Gupta. Load and store reuse using register file contents. In *Proceedings of the 15th International Conference on Supercomputing*, pages 289–302, 2001.

[23] S. J. Patel and S. S. Lumetta. rePLay: a hardware framework for dynamic optimization. In *IEEE, Transactions on Computers*, pages 50(6):300–318, June 2001.

[24] V. Petric, A. Bracy, and A. Roth. Three extensions to register integration. In *Proceedings of the 35th Annual International Symposium on Microarchitecture*, pages 37–47, 2002.

[25] D. Ponomarev, G. Kucuk, O. Ergin, and K. Ghose. Isolating short-lived operands for energy reduction. *IEEE Transactions on Computers*, 53(6):697–709, June 2004.