Specialized Dynamic Optimizations for High-Performance Energy-Efficient Microarchitecture

Yoav Almog, Roni Rosner, Naftali Schwartz and Ari Schmorak
Microprocessor Research
Intel Labs, Haifa, Israel
{ yoav.almog, roni.rosner, naftali.schwartz, ari.schmorak }@intel.com

Abstract

We study several major characteristics of dynamic optimization within the PARROT power-aware, trace-cache-based microarchitectural framework. We investigate the benefit of providing optimizations which although tightly coupled with the microarchitecture in substance are decoupled in time.

The tight coupling in substance provides the potential for tailoring optimizations for microarchitecture in a manner impossible or impractical not only for traditional static compilers but even for a JIT. We show that the contribution of common, generic optimizations to processor performance and energy efficiency may be more than doubled by creating a more intimate correlation between hardware specifics and the optimizer. In particular, dynamic optimizations can profit greatly from hardware supporting fused and SIMDified operations.

At the same time, the decoupling in time allows optimizations to be arbitrarily aggressive without significant performance loss. We demonstrate that requiring up to 512 repetitions before a trace is optimized sacrifices almost no performance or efficiency as compared with lower thresholds. These results confirm the feasibility of energy efficient hardware implementation of an aggressive optimizer.

1. Introduction

In this paper we study hardware-based dynamic optimizations within power-aware microarchitectures for high-performance, general-purpose processors. The essential challenge within this domain is the increasingly poor scaling of performance with power consumption. The PARROT (the Power-Aware aRchitecture Running Optimized Traces) microarchitecture proposes a trace-cache-based, decoupled mechanism for handling the frequently executed code – as initially reported in [29]. Within this microarchitectural framework we focus on the performance and energy-saving potential of dynamic optimizations performed on the most frequent traces in program execution.

Identifying frequently executed code sections for optimization has been applied in the software-based schemes reported in [15][7][1][10][2]. More recently, similar methods were suggested for hardware-based systems [17][18][23][22][8][27]. The various proposals differ in the methodology and resources used for detecting the hot paths, the structure and address space used for storing them, and the timing and resources used for optimization.

The PARROT system employs ***decoupled dynamic optimizations with memoization***. By ***decoupling*** our optimizer from execution, we can allow it to run numerous cycles without impeding execution progress. The hardware-based ***dynamic*** nature of our optimizations capitalizes on the advantages of microarchitectural-level optimizations over static optimizations provided by classical compilers. Even targeted JIT compilers may not be able to fully exploit the internal microarchitectural context of execution. Finally, ***memoization*** helps the microarchitecture avoid performing identical optimizations arbitrarily often.

PARROT exploits the dichotomy between frequent (or hot) and infrequent (or cold) code in hardware for the benefit of both processor performance and power awareness. The PARROT microarchitecture is designed to effectively identify the most frequent sequences of program code, aggressively optimize them once, and then efficiently execute them many times. Trace selection and filtering are used to identify the hot code, a dynamic optimizer is employed to optimize it, and a trace cache is used to store traces for repeated execution. Gradual construction of traces, pipeline decoupling, and specific trace optimizations are key factors for power awareness. We may in fact be willing to limit the hardware dedicated to the cold part of the code for a small price in performance. In return, we may be able to budget more aggressive hardware to improve performance/power tradeoffs for the dominant hot segments of the code.

We study the contribution of dynamic optimizations to processor performance and energy savings within two PARROT microarchitectural configurations: a ***narrow***

Proceedings of the International Symposium on Code Generation and Optimization (CGO 2004)
0-7695-2102-9/04 $20.00 © 2004 IEEE

configuration, characterized by an ordinary 4-wide execution pipeline and modest-size trace cache; and a more aggressive ***wide*** configuration, characterized by an 8-wide execution pipeline and a large trace-cache. Both configurations are tuned to resemble modern OOO processors designed for both high-performance and power-awareness. The studies are conducted with a variety of applications compiled for the IA32 Instruction Set Architecture (ISA).

By employing additional filtering on the hot traces that are actually optimized, we investigate the proportion of the frequent code on which dynamic optimizations have a significant impact. We demonstrate that it is sufficient to optimize a trace only after several hundreds of repetitions in order to retain almost all of the performance and energy-saving impact. Such infrequent application of the optimizations increases the freedom available for the hardware design of an optimizer and allows for a relaxed, power efficient design. Moreover, it enables the use of long-running and aggressive optimizations.

Focusing on this set of "blazing" traces, we compare the contributions of several major classes of optimizations. We measure the overall improvement in processor performance and energy consumption over the corresponding PARROT configurations with dynamic optimizations disabled. The contribution of dynamic optimizations can yield an average IPC improvement of 14-17% as well as 6-11% savings in energy, on the narrow and wide models, respectively. More than half of the IPC improvement and energy savings are due to hardware-specific optimizations. In particular, equipping the execution hardware with more powerful functional units enables specialized optimizations such as micro-op fusion and SIMDification. Further improvements may be attributed to micro-op rescheduling tailored to the available hardware and partial renaming for energy savings.

The rePlay system [22], although targeting performance issues, has much in common with PARROT techniques. PARROT and rePlay share the dual front-end, the decoupled, post-retirement construction of traces, and dynamic optimization of traces stored in a trace-cache. To promote power awareness, PARROT proposes a finer decoupling of trace construction based on gradual filtering in order to improve controllability of competing design metrics. PARROT's trace construction criteria are mostly static, enabling better adaptability to program structure. A good example is the handling of loops: by cutting loops at iteration boundaries, the PARROT microarchitecture prevents redundancy in the trace cache while still allowing loop unrolling. In contrast, the dynamic selection criteria of rePlay are in better synergy with the trace prediction mechanism. Our results complement and strengthen the rePlay study [31] showing the significant contribution of dynamic optimizations to IA32-based processors.

PARROT indeed goes beyond rePlay optimization scope by introducing core-specific optimizations which heavily exploit their integration into the hardware.

The paper is organized as follows. Section 2 describes the PARROT concept and microarchitecture, setting the stage for the rest of the paper. Section 3 details the optimization framework. Section 4 describes the simulation framework and defines the microarchitectural models compared in the current study. Section 5 presents the blazing filtering results, while Section 6 details the impact of major optimization classes. Finally, Section 7 concludes with a summary and ideas for future studies.

2. Parrot Microarchitectural Framework

The PARROT microarchitectural framework provides the enabling infrastructure for our research in trace filtering and dynamic optimizations. It is based on the following observations:

- The working set of a program is relatively small.
- Small segments of code which are repeatedly executed ("hot-traces") usually cover most of the program's working set.
- Hot traces, unlike other, less frequently executed code sections, are regular and predictable, and consequently exhibit higher potential for ILP extraction.

The PARROT concept suggests basing the development of high performance power-aware system on an ***asymmetric decoupling*** of the processor pipelines, as seen in Figure 2.1 (a slightly different decoupling concept is proposed in [3]). The left-hand and right-hand sides of the figure are responsible for executing the cold and hot portions of the code, respectively. Although the front-end is duplicated, the execution resources are shared between the hot and cold subsystems.

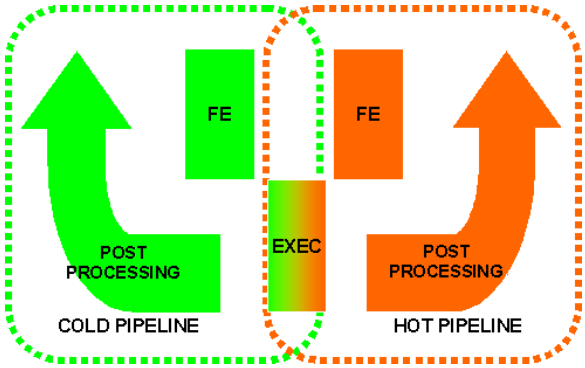

Figure 2.1 Schematic PARROT µarch

The cold and hot subsystems have a similar high level structure, with each being comprised of foreground and background components operating in parallel. The foreground components include the front-end and execution pipeline. The background components post-process the instruction flow out of the foreground pipeline making "off critical path" decisions such as when to move from the cold subsystem to the hot subsystem and when to apply optimizations.

Previous research [27][13] indicates that a trace cache can be very efficient in handling hot code, provided this code has been sufficiently well identified. (This is especially true in Intel's IA32 ISA which features variable length instructions.) Thus, the cold pipeline is fed with instructions fetched from an instruction cache, while the hot pipeline is fed with traces fetched from a trace cache. Both power-awareness and trace-cache effectiveness considerations limit trace construction and trace-cache insertion to frequently executed code sections. Thus, PARROT ***gradually*** applies dynamic optimizations — the hotter the trace is, the more aggressive power-aware optimizations are applied.

Decoupled dynamic optimizations with memoization have several advantages. ***Decoupling*** these optimizations from the foreground pipeline allows for more aggressive optimizations than the on-the-fly optimizations that can be performed within a standard execution pipeline. ***Dynamic*** information, most notably dynamic branch resolution, enables optimizations that are impossible for a static compiler. Finally, ***memoization*** within the trace-cache of decoded and optimized traces greatly enhances both performance and power savings.

Hardware-based optimizations have the advantage of being well integrated with the microarchitecture. Furthermore, microarchitectural level optimizations attain a high degree of ***architectural transparency***. The hardware is capable of optimizing legacy code, exploiting new microarchitectural features without the need for recompilation.

2.1. Traces and Trace-Selection

An execution ***trace*** is a sequence of operations representing a continuous segment of the dynamic flow (execution) of a program. Traces may contain execution beyond control-transfer instructions (CTIs), and so a trace may extend over several basic blocks.

In the current study we consider ***decoded atomic traces***. These traces contain ***decoded*** micro-operations (uops) and enable reuse of decode activity, thus saving energy [27] (decoded traces are of special value for IA32). Traces are constructed from the original uops in program order, but may later be optimized, resulting in an out-of-order, different, generally shorter sequence of uops.

Atomic traces are single-entry single-exit blocks [30]. Although atomic trace semantics requires a relatively complicated recovery mechanism and longer recovery time for the case of misprediction, it enables more aggressive optimizations, including uop reordering and elimination and branch promotion [22][23] and may efficiently utilize advanced trace prediction techniques such as those proposed in [12].

Trace selection is the activity of deciding which points in the dynamic instruction stream should be designated as trace start and end points. In the current study we apply the following selection criteria:

- Trace capacity is capped at 64 uops.
- With the exception of extremely large basic blocks, traces always terminate on CTIs.
- All ***indirect jumps*** and software exceptions terminate basic-blocks, except RETURN instructions. In addition, ***taken backward*** branches terminate a trace.
- RETURN instructions terminate traces only if they exit the outermost procedure context already encountered in the current trace.
- If two or more consecutive traces are identical, they are joined into a single trace, until the capacity limit is reached. This criterion, together with the taken-backwards termination condition on traces, achieves the effects of ***explicit loop unrolling***, an enabler for other optimizations.

With these criteria, unique ***trace identifiers*** (**TIDs**) can be compacted into a single address and a sequence of branch directions (taken/not taken). The only indirect CTI in this construction is a RETURN, but since its calling context is already part of the trace, its target address is implicitly available.

2.2. Microarchitecture

The background phase of the cold subsystem identifies frequent IA32 instruction sequences and captures them as traces in the trace cache. It is composed of TID selection, TID hot-filtering and finally trace-construction and insertion into the trace-cache. Since all committed instructions enter the TID selection phase, continuous training of both trace predictor and hot filter is assured. Nevertheless, only those TIDs that pass the hot-filter continue to the trace construction stage. The background phase of the hot subsystem identifies the most frequent (blazing) traces, optimizes them and finally inserts them back into the trace cache. Post processing is gradually performed, so the longer a trace is used the more aggressive optimizations are applied to it.

Two predictors are employed: A branch predictor predicts the next cache line designated to be fetched from the instruction cache for execution on the cold pipeline. Simultaneously, a trace predictor predicts the TID of the next trace designated to be fetched from the trace cache and executed on the hot pipeline. Each predictor is based on a global history register (GHR). The GHR is updated for each CTI being executed. Both predictors support speculative update upon fetch and real update on commit.

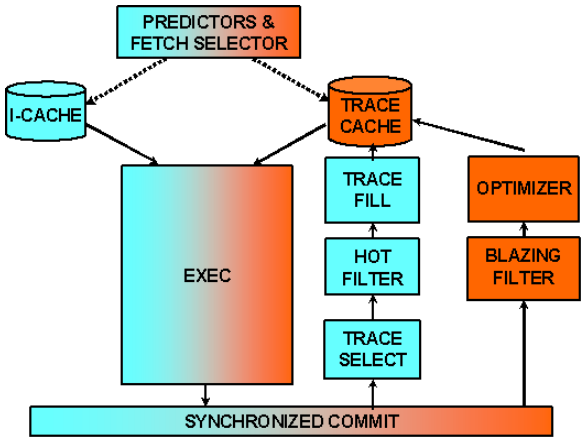

Figure 2.2

Figure 2.3 PARROT main μarch components

The fetch-selector chooses between the execution pipelines by consulting both the branch predictor and the trace predictor. When the trace predictor is successful in making a next TID prediction and a trace is successfully fetched from the trace-cache it is executed in the hot pipeline. Otherwise, cold pipeline execution is commenced using the result of the branch predictor.

For post-processing cold instructions, PARROT employs a non-speculative TID/trace build scheme. Cold committed instructions are collected as long as all encountered CTIs satisfy the trace selection criteria (see Section 2.1). When a termination condition is reached, a new TID, generated from the collected CTIs, is used to train the trace predictor. If the TID is subsequently identified as frequent (see below), the collected micro-ops are used to construct an executable trace that can be inserted into the trace cache.

In order to identify the frequently executed instruction sequences, PARROT gradually employs two filtering mechanism: the hot filter, which is used for selecting frequent TIDs from among those constructed on the cold pipeline, and the blazing filter, which is used for selecting the most frequent TIDs from among those executed on the hot pipeline. Both filters are small caches that retain access counters for each TID. Each trace execution increments the corresponding counter. Once the hot filter threshold is reached, the trace is constructed and inserted into the trace cache. When the blazing filter threshold is reached, the executed trace is optimized and written back to the trace cache, replacing the original.

3. Dynamic Optimizations

The PARROT optimizer is capable of performing many different optimizations effectively. The infrastructure maintains a symbol table and a static dependency graph, both of which may be implemented as fast hardware arrays. While optimizing, the symbol table and the dependency graphs are updated incrementally with ongoing transformations. Optimizations are carried out in several passes, with each building upon the results of prior passes. Optimizations provide the following benefits:

- **Code reduction**: eliminates uops, saving execution time and reducing pressure on the reservation station and other internal buffers.
- **Dependency elimination**: reduces pressure on the register file, and improves ILP.
- **Partial renaming**: saves renaming effort using virtualization of intra-trace registers and pre-identification of live-ins/live-outs.
- **Improved scheduling**: improves execution time by reordering uops to reduce average wait-time of dependent uops.

As noted above, optimizations can be classified as either ***generic*** or ***core-specific***.

Generic optimizations are independent of the underlying execution core, but rely heavily on the specific semantics of IA32 instructions. These optimizations extend the scope of classical compiler optimizations by operating across basic block boundaries (atomicity is assured by replacing conditional jumps with assert instructions). Consequently, they are more effective at breaking dependencies, reducing code size and enabling advanced optimizations. Generic optimizations include:

- **Logic simplifications**: **A ^ A → 0; A & A → A; A | 0 → A**.
- **Arithmetic simplifications**: constant folding, condition propagation and address manipulation. These optimizations enable further memory simplifications and dependency breaking, particularly for unrolled loops.
- **Memory simplification**: shadowed store elimination, shadowed load eliminations and store forwarding. These optimizations reduce memory traffic and shorten dependencies, and may also eliminate uops.
- **Data flow simplification**: dead code elimination and move propagation. These optimizations reduce code size and simplify dependencies.

Core-specific optimizations are tailored to a specific hardware implementation and exploit internal micro-architectural features to go beyond generic optimizations. It is important to note that the generic optimizations must precede the core-specific optimizations. Core-specific optimizations include:

- **Partial renaming:** Partial renaming is targeted at power/energy reduction as well as performance and is an enabler for all other optimizations. The optimization transforms constituent uops to SSA (static single assignment), constructing def-use chains and linking them into the symbol table. Since registers are partially renamed to consecutive static-virtual registers, only a simple additional mapping is needed to adjust to free locations in the register pool. This in turn reduces runtime overhead.

- **Fusion:** Fusion is targeted at reduction of dependencies, register-file pressure and code. The idea is to collapse local dependencies associated with consecutive uops along a dependency chain in which all intermediate values are either constant or produced internally. Such chains may be implemented as one fused operation in hardware, eliminating all intermediate register values. We have found the following sequences especially attractive for fusion (assert instructions are the result of COMPARE / CONDITIONAL-JUMP instruction pairs):

  sub , assert → sub_&_assert

  and, assert → and_&_assert

  shl, add → shl_&_add

- **SIMDification:** SIMDification is targeted at reduction of code, register-file pressure, memory traffic, and execution latency. The idea is to merge operations that may be scheduled together for execution and require similar execution resources (such as registers and functional units). The merged uops are replaced with a new SIMDified operation. It is important not to combine operations in a manner which will increase dependencies or complicate the detection of SIMDfiable uops. Therefore, we constrain SIMDification to combine only operations which have identical or constant inputs and additionally SIMDfied uops should share the same tree-height in trace dependency graph. Since the SIMDified operation contains only one instance of each non-const input, many redundant register-file accesses are eliminated.

In case the SIMDified operation is a memory operation, all merged operations are constrained to be of fixed stride relative to each other. This helps reducing memory traffic by compressing consecutive memory requests. Since SIMDified memory operations may partially miss in the cache, this optimization may at times hurt performance, and must be applied with care. Our studies indicate that the following sequences are attractive for SIMDification:

  Fus_sub_assert, Fus_sub_assert → simd_sub_assert

  load(base,4), load(base+4,4) → load(base, 8)

- **Pre-scheduling** Pre-scheduling reorders uops to improve overall execution latency. It uses a heuristic in which operations that have longer critical path are positioned earlier in the trace. The critical path of an operation is the longest distance (estimated execution cycles) of any path of dependent operations in the dependency graph. This heuristic is suitable for applications that make heavy use of long latency operations.

3.1. Hardware implementation

Although space considerations do not permit us to fully elaborate hardware implementation details of all optimizations, we sketch some microarchitectural techniques employed for the relatively complex SIMDification. Recall that SIMDification is limited to uops that share the same tree-height in the trace dependency graph. Since the dependency graphs changes as a result of many optimizations, it is important to perform SIMDification only after the simpler optimizations have completed. For example, the generic optimizations may break dependencies between the iterations of an unrolled loop, paving the way for cross-iteration SIMDification (see Table 3.1 below).

A small set-associative cache holds pointers to candidate uops for SIMDification. The sets are indexed by the tree height of uops, and entries are tagged by additional SIMD-enablers, including operation type and register number. Additional information, such as dependency between uops is maintained in a standard dependency matrix.

Each uop in turn is checked against matching candidates in the set corresponding to its tree height. A tag match indicates that the two uops can be grouped into a legal SIMD. If no conflicting memory dependencies are found, the SIMDification replaces the original uops in the cache for further potential matching. When there is no match, the current uop itself is inserted to the SIMD cache.

3.2. Examples

We present a few code examples taken from blazing traces of actual applications. Table 3.1 presents a real world example trace (taken from MS word) of a loop with 3 unrolled iterations. The first column of the table presents the original IA32 instructions of the original trace, while the 2nd and 3rd columns present the uops of the decoded trace, before and after basic transformations, respectively. Notice that the basic transformations replace all conditional branches with control flow assertions: original uops 3, 6, 13, 16, 23 and 26 are transformed into uops 3, 6, 12, 15, 21 and 24, respectively. Interior direct branches 9 and 19 are eliminated. In addition, all registers

are transformed to SSA form: internally-used architectural register esi in uop 27 is virtualized into v_b(6) in corresponding transformed uop 25. Live-ins/live-outs are appropriately designated: register esi in uop 28 is transformed into lo_esi in uop 26 (prefixes "li_", "lo_" designate live/in and live-out registers, respectively). The 4th column shows the output of the optimizer, in which 18 operations are eliminated (64%), 20 register definitions are avoided (67%) and critical path length is reduced from 7 to 2 (71%). In addition, 3 LOAD operations as well as 3 SUB operations are combined as SIMD: original uops 4, 13 and 22 are optimized to uop 0, while many CMP-ASSERT operations are fused: 23 and 24 are fused to uop 4; some uops are shuffled by pre-scheduling. Note that these transformations are enabled with the help of many prior code simplifications.

IA32 Instructions	Original PARROT uops	Uops after basic-transformations	Uops after optimizations
0. ***mov*** eax ← edi	**0.** eax ← ***mov*** (edi)	**0.** v_a(0) ← ***move*** (li_edi)	**0.** v_c(0),v_c(1), lo_cx
1. ***dec*** edi	**1.** edi, eflags ← ***sub*** (edi, 1)	**1.** v_b(0), v_flags(0) ← ***sub*** (li_edi, 1)	←***3Xload***(li_ds,li_esi)
2. ***testl*** eax, eax	**2.** eflags ←***and*** (eax, eax)	**2.** v_flags(1) ← ***and*** (v_a(0), v_a(0))	**1.** v_b(0),v_a(1),v_b(3),lo_eax, lo_edi
3. ***je*** +0x14	**3.** ***cond_jmp***(e,eflags)	**3.** ***assert_cond*** (v_flags.z(1), e, ntaken)	←***3Xsub***(li_edi,1, li_edi,2, li_edi, 3)
4. ***mov*** cx ← [esi]	**4.** cx ← ***load*** (ds, esi)	**4.** v_c(0) ← ***load***(li_ds,li_esi)	***2. jmp*** (0x3004d8a5)
5. ***cmp*** cx, 256	**5.** eflags ← ***sub*** (cx, 256)	**5.** v_flags(2) ← ***sub*** (v_c(0), 256)	**3.** lo_esi ←***add***(li_esi, 0x6)
6. ***ja*** 0x2c645e	**6.** ***cond_jmp*** (nbe, eflags)	**6.** ***assert_cond***(v_flags.cz(2),nbe, ntaken)	**4.** lo_eflags
7. ***inc*** esi	**7.** esi, eflags ← ***add*** (esi, 1)	**7.** v_b(1), v_flags(3) ← ***add*** (li_esi, 1)	←***sub&assert***(lo_cx,256,nbe,ntaken)
8. ***inc*** esi	**8.** esi, eflags ← ***add*** (esi, 1)	**8.** v_b(2), v_flags(4) ← ***add*** (v_b(1), 1)	***5. and&assert***(v_b(3),v_b(3), e, ntaken)
9. ***jmp*** -23	**9.** ***jmp***(0x3004d8a5)		***6. sub&assert***(v_c(1), 256, nbe, ntaken)
---	---	---	***7. and&assert***(v_b(0), v_b(0),e, ntaken)
10. ***mov*** eax ← edi	**10.** eax ← ***mov*** (edi)	**9.** v_a(1) ← ***move*** (v_b(0))	***8. sub&assert***(v_c(0), 256, nbe, ntaken)
11. ***dec*** edi	**11.** edi, eflags ←***sub*** (edi, 1)	**10.** v_b(3), v_flags(5) ← ***sub*** (v_b(0), 1)	***9. and&assert***(li_edi, li_edi, e, ntaken)
12. ***testl*** eax, eax	**12.** eflags ←***and*** (eax, eax)	**11.** v_flags(6) ← ***and*** (v_a(1), v_a(1))	
13. ***je*** +0x14	**13.** ***cond_jmp***(e,eflags)	**12.** ***assert_cond*** (v_flags.z(6),e, ntaken)	
14. ***mov*** cx ← [esi]	**14.** cx ← ***load*** (ds, esi)	**13.** v_c(1) ← ***load***(li_ds,v_b(2))	
15. ***cmp*** cx, 256	**15.** eflags ← ***sub*** (cx, 256)	**14.** v_flags(7) ← ***sub*** (v_c(1), 256)	
16. ***ja*** 0x2c645e	**16.** ***cond_jmp*** (nbe, eflags)	**15.** ***assert_cond***(v_flags.cz(7),nbe, ntaken)	
17. ***inc*** esi	**17.** esi, eflags ← ***add*** (esi, 1)	**16.** v_b(4), v_flags(8) ←***add*** (v_b(2), 1)	
18. ***inc*** esi	**18.** esi, eflags ← ***add*** (esi, 1)	**17.** v_b(5), v_flags(9) ← ***add*** (v_b(4), 1)	
19. ***jmp*** -23	**19.** ***jmp***(0x3004d8a5)		
---	---	---	
20. mov eax ← edi	**20.** eax ← ***mov*** (edi)	**18.** lo_eax ← ***move*** (v_b(3))	
21. dec edi	**21.** edi, eflags ←***sub*** (edi, 1)	**19.** lo_edi,v_flags(10)←***sub***(v_b(3), 1)	
22. testl eax, eax	**22.** eflags ←***and*** (eax, eax)	**20.** v_flags(11)←***and***(lo_eax,lo_eax)	
23. je +0x14	**23.** ***cond_jmp***(e,eflags)	**21.** ***assert_cond***(v_flags.z(11), e, ntaken)	
24. mov cx ← [esi]	**24.** cx ← ***load*** (ds, esi)	**22.** lo_cx ←***load***(li_ds,v_b(5))	
25. cmp cx, 256	**25.** eflags ← ***sub*** (cx, 256)	**23.** lo_eflags ←***sub***(lo_cx, 256)	
26. ja 0x2c645e	**26.** ***cond_jmp*** (nbe, eflags)	**24.** ***assert_cond***(lo_eflags(65),nbe, ntaken)	
27. inc esi	**27.** esi, eflags ← ***add*** (esi, 1)	**25.** v_b(6),v_flags(12) ←***add*** (v_b(5), 1)	
28. inc esi	**28.** esi, eflags ← ***add*** (esi, 1)	**26.** lo_esi,v_flags(13)←***add***(v_b(6), 1)	
29. jmp -23	**29.** ***jmp***(0x3004d8a5)	**27.** ***jmp*** (0x3004d8a5)	

Table 3.1 Optimized trace example from SysMark 2000 Microsoft word

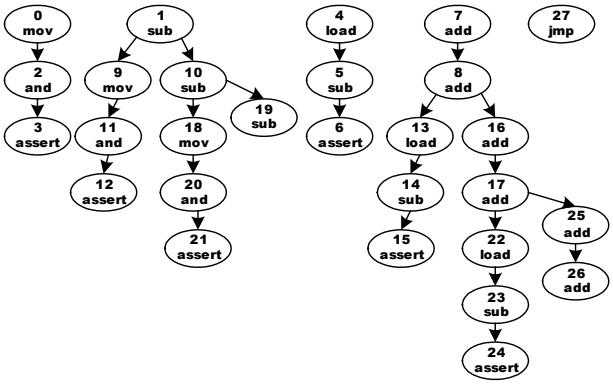

Figure 3.1 Trace dependency graph after basic transforms

The corresponding reduction in dependencies and critical path length can be observed by comparing Figure 3.1, the dependency graph after basic transformations, with Figure 3.2, the dependency graph after optimizations.

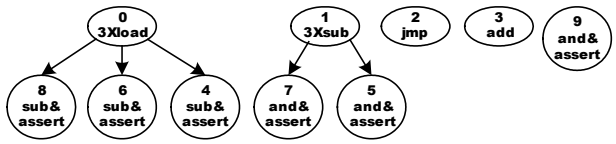

Figure 3.2 Trace dependency graph after optimizations

Next example, presented in Table 3.2, is a trace from gcc containing a loop with 2 unrolled iterations. As above, columns 1, 2 and 3 of the figure present the original trace, decoded trace and transformed trace, respec-

tively. The 4th column shows the output of the optimizer, in which 9 operations are eliminated (56%), 4 register definitions are avoided (40%) and critical path length is reduced from 7 to 5 (40%). In addition, a store operation is avoided, the original uops 1 and 2 are eliminated, and the load operation of uop 8 is replaced by a simple move.

IA32 Instructions	Original PARROT uops	Uops after basic-transformations	Uops after optimizations
0. mov edx ← [esp+4]	***0.*** edx ← ***load*** (ss, esp, 4)	***0.*** v_a(0) ← ***load*** (li_ss, li_esp, 4)	***0.*** v_a(0), lo_esi, lo_ecx ← ***load*** (li_ss, li_esp, 4)
1. mov [esp+4] ← esi	***1. store_data*** (esi)	***1. store_data*** (li_esi)	***1.*** lo_edx, lo_edi ← ***move*** (li_esi)
2. mov esi ← edi	***2. store_address*** (ss, esp, 4)	***2. store_address*** (li_ss, li_esp, 4)	***2.*** lo_eflags ← ***and*** (lo_edx, lo_edx)
3. mov ecx ← edi	***3.*** esi ← ***move*** (edi)	***3.*** v_b(0) ← ***move*** (li_edi)	***3. cond_jmp***(NZ, lo_eflags)
4. mov edi ← edx	***4.*** ecx ← ***move*** (edi)	***4.*** v_a(1) ← ***move*** (li_edi)	***4. store_address*** (li_ss, li_esp, 4)
5. test edx ← edx	***5.*** edi ← ***move*** (edx)	***5.*** v_b(1) ← ***move*** (v_a(0))	***5. store_data*** (li_edi, 2)
6. jnz .-16	***6.*** eflags ← ***and*** (edx, edx)	***6.*** v_flags(0) ← ***and*** (v_a(0), v_a(0))	***6. fus_and_assert***(v_a(0),v_a(0),nz,taken)
	7. cond_jmp(nz, eflags)	***7. assert_cond*** (v_flags(0), nz, taken)	
----------	----------	----------	
7. mov edx ← [esp+4]	***8.*** edx ← ***load*** (ss, esp, 4)	***8.*** lo_edx ← ***load*** (li_ss, li_esp, 4)	
8. mov [esp+4] ← esi	***9. store_data*** (esi)	***9. store_data*** (v_b(0))	
9. mov esi ← edi	***10.store_address*** (ss, esp, 4)	***10. store_address*** (li_ss, li_esp, 4)	
10. mov ecx ← edi	***11.*** esi ← ***move*** (edi)	***11.*** lo_esi ← ***move*** (v_b(1))	
11. mov edi ← edx	***12.*** ecx ← ***move*** (edi)	***12.*** lo_ecx ← ***move*** (v_b(1))	
12. test edx ← edx	***13.*** edi ← ***move*** (edx)	***13.*** lo_edi ← ***move*** (lo_edx)	
13. jnz .-16	***14.*** eflags ← ***and*** (edx, edx)	***14.*** lo_eflags←***and*** (lo_edx, lo_edx)	
	15. cond_jmp(nz, eflags)	***15. cond_jmp*** (nz, lo_eflags)	

Table 3.2 Optimized trace example from SpecInt 2000 gcc

3.3. Unrolling Degree

To obtain some measure of the effects of different unrolling degrees on trace execution efficiency we plotted various efficiency metrics against the number of loop iterations unrolled in the gcc example above.

Figure 3.3 demonstrates the non-linear optimization impact across different unrolling degrees. In this example, optimization effectiveness rises across higher unrolling degrees. However, maximum effectiveness is attained with 6-fold unrolling, achieving almost 80% reduction. Note that about half of the impact is obtained with only two loop iterations.

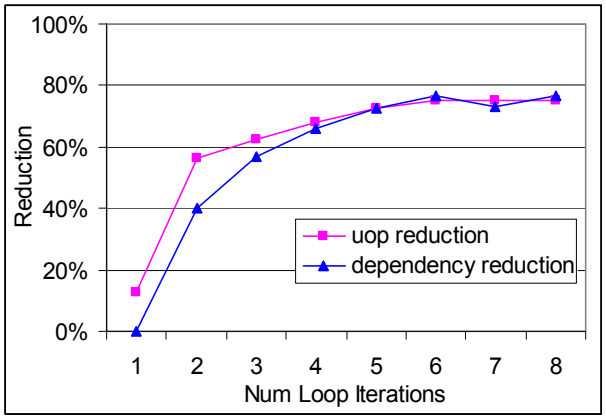

Figure 3.3 Impact of loop unrolling on uop and dependency reduction in gcc trace example

4. Simulation Framework

We employ an in-house proprietary simulation environment as a modeling and research vehicle for the PARROT microarchitecture. The simulators are designed with maximum flexibility and configurability in order to enable comparative study of the diverse set of microarchitectural alternatives detailed in Section 4.3, on the diverse set of benchmark applications listed in 4.2.

The simulators are trace-driven; they simulate execution traces of applications compiled for the IA32 architecture. They implement all the components of the PARROT microarchitecture, including the less traditional optimizer, and all of the optimizations and precomputations described above.

4.1. Energy Simulation

For energy simulation we employed a WATTCH-like approach, using accurate and up-to-date data extracted from real processors for the internal components, ensuring the relevancy of our results.

The power-modeling infrastructure is based on Intel propriety tools. These include formulas for small functional blocks, each of which was closely correlated to the corresponding hardware implemented on recent technology. The formulas are composed of arithmetic expressions involving parameters which stand for dynamic events (i.e. counters). A formula is designed to predict the dynamic energy consumption of the block, and its sub-formulas cover both active and idle power.

The methodology we developed allows us to use the correlated formulas as building blocks, and compose them into larger formulas describing the energy consumption of higher-level units in our micro-architecture. In this composition process we utilize three mechanisms: ***instantiation***, ***mapping*** and ***scaling***. For each unit in the microarchitectural model we ***instantiate*** a set of formulas that best fit the functionality and energy behavior of that unit. For each formula we provide a ***mapping*** of its original parameters into actual event-counters of the simulated model. In addition, each formula is potentially the subject of ***scaling*** representing uniform size or activity enlargements over the original block. There are separate scaling factors for active and for idle power, representing extensions such as increased number of ports, increased number of cache entries, etc.

Consequently, power modeling for conventional units is relatively straightforward, whereas the power modeling for new units which have no similar original formulas (e.g., the optimizer or the trace constructor) requires more work. For example, the power model of the optimizer employs instances of buffers and tables taken from the ROB, rename and scheduling stages of the pipeline, scaled down to accommodate a single trace, and with some combinations of significant optimizer events mapped to the original execution events.

Leakage is derived from the dynamic power under some simplifying assumptions. We assume uniform leakage 1) in space over two coarse component types, the processor core and the level-2 cache, and 2) in time, modeling a consistently high temperature.

To emulate the high temperature, we choose the application with highest average dynamic-power *PMAX* of the base OOO model. This turns out to be swim of the SpecFP suite (see below). For a component area *A*, we define the uniform leakage power as *A * PMAX * T*, where *T* is a technology constant. We use technology constants of 5% for each MByte of level 2 cache and 40% for the standard core. These fairly large constants are used in accordance with the technological trend of increasing leakage. Thus, for a model with *M* Mbytes of L2 cache and a core of area *K* times the standard OOO core, the total leakage energy *LE* of an application running for *CYC* cycles is modeled by the formula

$$LE = PMAX * (0.05 * M + 0.4 * K) * CYC$$

This infrastructure produces energy estimations for the different micro-architecture models and their components, usable for global trade-off analysis. For more details on the simulation methodology and comparative results (e.g. breakdown of energy/power consumption of different microarchitectural components) we refer the reader to [29].

4.2. Benchmarks

Our benchmark suite covers a wide range of optimized application traces, 30 or 100 million instructions each. The 30 application runs can be classified as follows:

- **SpecInt** 2000: bzip, crafty, gap, gcc, gzip, parser, perlbmk, vortex (30M).
- **SpecFP** 2000: art, facerec, fma3d, lucas, mesa, sixtrack, swim, wupwise (30M).
- **Office** / Windows applications from SysMark 2000: office, powerpoint, virusscan, word (100M).
- **Multimedia**: flash (from SysMark-2000, 100M), Dragon, lightwave, quakeIII, 3DsMax (light, raster and geom), Flask-MPEG4 (custom Multimedia traces, 30M).
- **DotNet**: image, phong (custom applications, 100M).

4.3. Models

To investigate the influence of our thresholded dynamic optimizations we configured the following models. First, using our most aggressive optimizations, we varied the number of trace repetitions which would trigger trace optimization from 8 to 16K, adding one datapoint to represent a threshold of infinity under which optimization would never be triggered. Table 4.1 summarizes our models for narrow and wide cores. The microarchitectural parameters used for these configurations are summarized in Table 4.2.

Threshold / Core	8	64	512	4096	16384	-
Narrow	N8	N64	N512	N4K	N16K	N-
Wide	W8	W64	W512	W4K	W16K	W-

Table 4.1 Configurations using different blazing thresholds

Using the optimum threshold indicated by running these models on our benchmarks, we then investigated the contribution of each of our optimizations in isolation to determine which are most crucial for improving power and performance. For these experiments, we divided our optimizations into a number of general classes:

- **G**: Generic (logic/flow/memory) simplifications
- **S**: Pre-Scheduling
- **F**: Uop Fusions
- **X**: SIMDifications

Parameter	N*	W*
Core area (relative to base OOO)	1.4	2.4
Predictor entries (branch + trace)	2K + 2K	4K + 4K
FHR length (branch + trace)	16 + 32	16 + 32
Icache	32KB	32KB
Tcache entries	128	512
FE pipeline width (cold + hot)	4 + 4	4 + 8
ROB entries	100	150
Sched. Window	32	50
Exec. Ports	4	8
EXEC pipeline width	4	8
Hot filter (entries, threshold)	1K, 24	1K, 8
Blazing filter (entries, threshold)	1K, *	1K, *
Max uops in trace	64	
Optimizations enabled	*	
Optimizer latency (non pipelined)	100 cycles	
L1 Dcache (size, latency)	64KB, 3	
L2 Ucache (size, latency)	2MB, 9	
Memory latency	120	
Line size (I and D)	64B	
All caches (I, D, T) associativity	8-way, LRU	

Table 4.2 µarch settings of narrow / wide models. The "*" denotes free parameters varying across configurations

Table 4.3 shows a variety of optimization configurations, ranging from none to most aggressive. Since the generic optimizations are enabling for all others, they were included with every optimizing configuration.

Optimizations	Narrow	Wide
none	N-	W-
G	N_G	W_G
G and F	N_GF	W_GF
G and X	N_GX	W_GX
G and S	N_GS	W_GS
G, F, X and S	N512	W512

Table 4.3 Optimization Configurations

4.4. Metrics

We employ a variety of metrics designated to evaluate and compare different aspects of the simulated models. For the overall processor performance we focus on the IPC, total energy and Cubic-MIPS-per-WATT (CMPW) measurements. IPC and total energy are useful for understanding the design tradeoffs assuming the same frequency and same voltage. The CMPW metric is instrumental in quantifying the design tradeoffs and power awareness of a processor assuming energy consumption could be always traded for performance using voltage or frequency scaling [4][32].

We also present metrics related to optimizations, such as optimized code ("blazing") coverage, uop reduction and optimizer efficiency (or utilization) and sensitivity.

5. Results: Blazing Filter Threshold

This section presents results on a range of thresholds for the blazing filter. Based on the observed trends, a threshold providing a reasonable power/performance tradeoff is chosen as a reference point against which the gains from various optimization configurations are compared with one another in Section 6 below.

5.1. Optimizations Impact

In Figure 5.1--Figure 5.3 we observe the impact of the blazing threshold on two major microarchitectural parameters, IPC and energy consumption, as well as its impact on a major parameter characterizing the impact of dynamic optimizations, namely uop-reduction. Although there is a clear growth trend in IPC gain and energy reduction when the threshold gets lower, this growth is not linear. In most cases, the observed gain diminishes significantly below a threshold of 512.

Considering the different benchmark classes, we observe that primarily Office, and to a leaser degree Multimedia and SpecInt applications obtain both performance and energy savings benefits from optimizations. The high uop reduction on SpecFP and .NET applications is not fully realized in performance since it is masked by high memory activity.

The wide models display consistently higher IPC improvement and energy savings than the narrow models. This can be attributed to the higher uop reduction and better availability of hardware resources, such as larger trace cache that can store more optimized traces.

For our purposes, the number of 512 iterations serves as a good ***constant*** threshold. Higher thresholds represent a slightly reduced benefit in overall energy and performance, but this threshold delivers efficiency while maintaining low sensitivity of overall performance and energy to optimization latency and energy consumption. Future studies may consider more sophisticated adaptive filtering techniques.

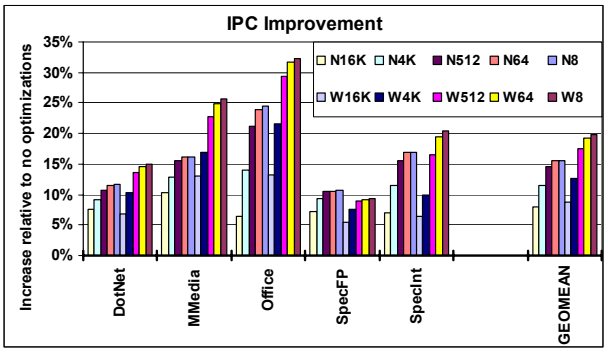

Figure 5.1 IPC improvement relative to base non-optimized models (N- and W-, respectively)

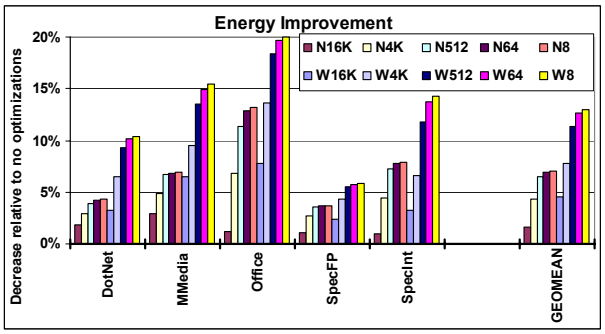

Figure 5.2 Energy improvement relative to base non-optimized models

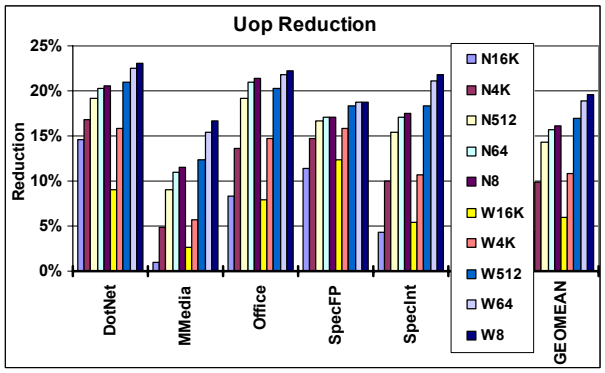

Figure 5.3 Uop-reduction, out of all committed uops in the non-optimized models

5.2. Optimizer Parameters

The next figures demonstrate the appropriateness of the 512 threshold with regard to finer characteristics of our dynamic optimization model.

The blazing coverage (of optimized code) is depicted in Figure 5.4. About 80% blazing coverage is achievable with our 512 threshold, whereas larger thresholds incur a sharp drop in coverage. Conversely, smaller thresholds exhibit lower utilization of the optimizer (Figure 5.5). The 512 threshold is sufficiently large to guarantee that any optimized trace is executed more than 3500 times in average on the wide models, and more than 450 times on the narrow ones.

In Figure 5.6 and Figure 5.7 we demonstrate that the full benefits of optimization with blazing threshold of 512 are obtained with an optimizer activated very infrequently: less than 50 times (8 times for wide models) every Million committed instructions, or at most once in 11000 cycles (60000 cycles of our wide models).

This low frequency of optimizer activation indicates low sensitivity to the actual time it takes to optimize a trace. It also hints at the feasibility of implementing a reasonably complex optimizer as part of a power-aware processor core.

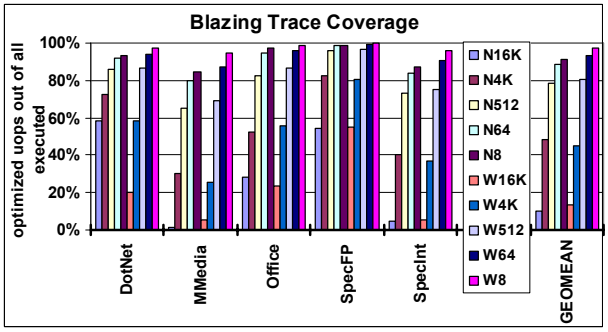

Figure 5.4 Coverage of uops from optimized traces out of all executed uops

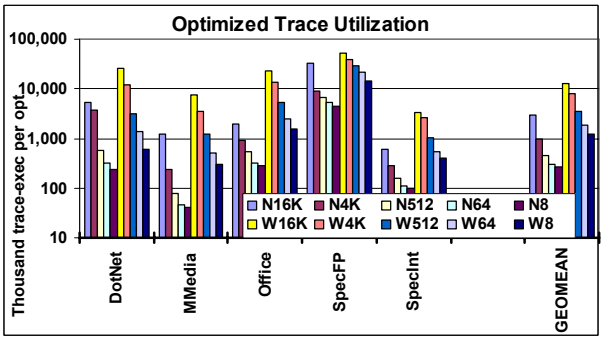

Figure 5.5 Utilization of optimizer, measured in thousands of executed optimized traces per optimization

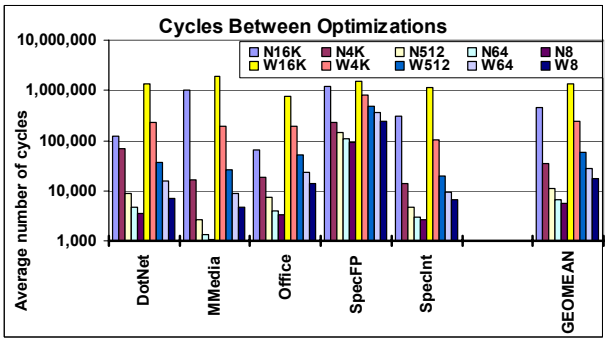

Figure 5.6 Average number of cycles between consecutive activations of the optimizer

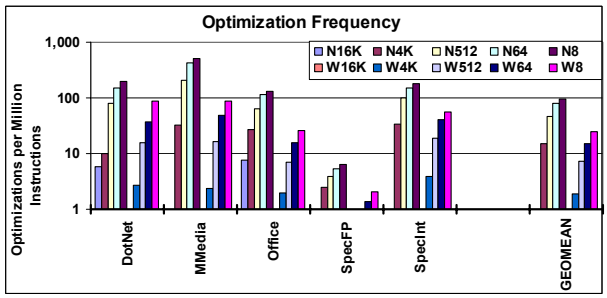

Figure 5.7 Frequency of activating the optimizer (per Million committed instructions)

6. Results: Optimizations Breakdown

In this section we analyze the contribution of each of our major optimization classes using our chosen blazing threshold of 512.

The figures below depict the IPC improvement obtained by several optimization classes implemented in the narrow and wide models, respectively. The presented combinations include the generic optimizations (G), the individual addition to G of each of the core-specific fusion (GF), SIMD (GX) and scheduling (GS), and full optimizations (with the 512 threshold). The results are shown for each of the benchmark groups.

The benefit of the core-specific optimizations is a major contribution on top of the generic ones. The IPC improvement (Figure 6.1) of the full optimizations more than doubles that of generic optimizations alone, from 7.4% to 14.6% on the narrow models and from 6.1% to 17.5% on the wide ones.

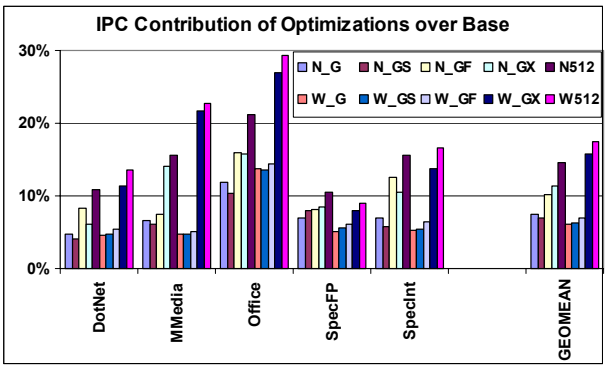

Figure 6.1 IPC improvement over the reference, non-optimized models N- and W-, respectively

At the same time, the energy advantage (Figure 6.2) of the full optimizations more than triples that of the generic ones, extending it from 1% to 6.5% on the narrow models and from 2.8% to 11.4% on the wide ones.

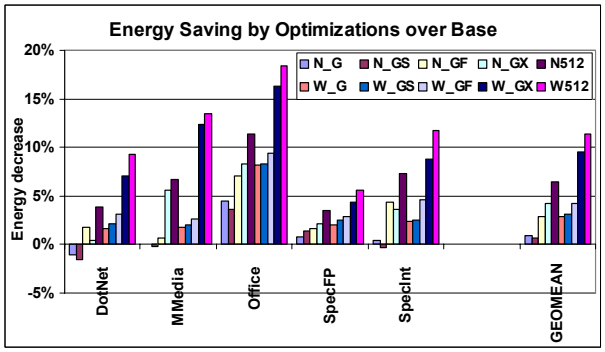

Figure 6.2 Energy improvement (decrease) over the reference, non-optimized models N- and W-, respectively

The energy results of some benchmarks are less conclusive when run on the narrow models. In order to summarize more conclusively the performance and energy tradeoffs, we produced the CMPW results for all benchmark groups. These are presented in Figure 6.3 and confirm the significance of dynamic optimizations in general and of core-specific optimizations in particular, on all models and with all benchmarks.

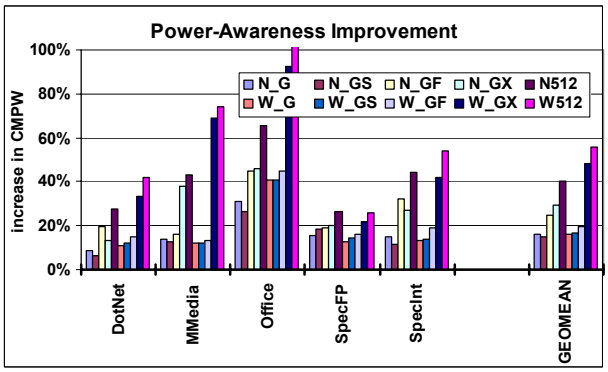

Figure 6.3 Overall improvement in power-awareness (CMPW) over N- or W-, respectively

7. Conclusions and Future Work

We have investigated the potential of a hardware implementation of a dynamic optimizer in a power-aware context. Our study has shown that optimizations which are tightly coupled to the microarchitecture in substance but not in time can provide a substantial benefit in both performance and energy consumption.

Our filtering experiments have demonstrated that cautious selection of our optimization candidates costs only a small reduction in the power/performance benefit of the optimizations. It also proves that considerable timing leeway is available and allows for a simpler-design, longer-latency and lower-power hardware optimizer implementation.

We have also shown that the available benefit corresponds to the original investment in optimization effort: more aggressive optimizations yield better results in both performance improvement and energy savings. At the same time, exposure of the hardware characteristics to the optimizer is essential for making significant improvements over those achievable by standard generic techniques.

This finding paves the way for further studies into the precise form a hardware optimizer should take. For example, we can take advantage of predicted utilization of the optimizer in order to customize our optimization strategy by individual trace. Expected utilization may be

estimable from characteristics of the trace itself, or may come from accumulated experience with the current workload, or both. This information may then be used for training more adaptive filters.

Although the PARROT system provides the capability of gradual optimization, the optimization grain currently studied is very coarse: either a trace is optimized or it is not. A natural extension is a system with a finer optimization grain, allowing us to fine-tune the filtering-guided optimization investment in a much more precise manner, according to the cost and benefit of the optimizations. Such strategies would improve our return on optimization effort.

The actual construction of compiler-style optimization circuitry is itself a fairly new field and further design research will be needed to indicate the best manner to proceed. These technologies may open a new dimension in power-aware computing.

References

[1] V. Bala, E. Duesterwald and S. Banerjia, "Transparent Dynamic Optimization: The Design and Implementation of Dynamo", TR HPL-1999-78, HP Labs.

[2] L. Baraz, T. Devor, O. Etzion, S. Goldenberg, A. Skaletsky, Y. Wang and Y. Zemach, "IA-32 Execution Layer: A Two Phase Dynamic Translator Designed to Support IA-32 Applications on Itanium-Based Systems", in *MICRO36*, Dec. 2003.

[3] B. Black and J.P. Shen, "Turboscalar: A High Frequency High IPC Microarchitecture", in *ISCA27*, June 2000.

[4] D.M. Brooks et al, "Power-Aware Microarchitecture: Design and Modeling Challenges for Next-Generation Microprocessors", *IEEE Micro*, 20(6):36-44, Nov/Dec. 2000.

[5] G. Cai, C.H. Lim and W.R. Daasch, "Thermal-Scheduling For Ultra Low Power Mobile Microprocessor", in *Proc. WCED'02*, 2002.

[6] T. M. Conte, K. N. Menezes, P. M. Mills and B. A. Patel, "Optimization of instruction fetch mechanisms for high issue rates",in *ISCA22*, Jun. 1995.

[7] K. Ebcioglu and E.R. Altman, "DAISY: Dynamic Compilation for 100% Architectural Compatibility", in *ISCA24*, pp. 26-37, 1997.

[8] B. Fahs, S. Bose, M. Crum, B. Slechta, F. Spadini, T. Tung, S.J. Patel and S.S. Lumetta, "Performance Characterization of a Hardware Mechanism for Dynamic Optimization", *MICRO34*, Dec. 2001.

[9] D. Friendly, S. Patel and Y. Patt, "Putting the Fill Unit to Work: Dynamic Optimizations for Trace Cache Microprocessors", in *MICRO31*, Nov. 1998.

[10] M. Gschwind, E.R. Altman, S. Sathaye, P. Ledak and D. Appenzeller, "Dynamic and Transparent Binary Translation", in *IEEE Computer Magazine* 33(3), pp. 54-59, 2000.

[11] Q. Jacobson, E. Rotenberg and J.E. Smith, "Path-Based Next Trace Prediction", in *MICRO30*, 1997.

[12] Q. Jacobson and J.E. Smith, "Trace Preconstruction", in *ISCA27*, pp. 37-46, May 2000.

[13] O. Kosyakovsky, A. Mendelson and A. Kolodny, "The Use of Profile-based Trace Classification for Improving the Power and Performance of Trace Cache Systems", in *4th Workshop on Feedback-Directed and Dynamic Optimization*, Dec. 2001.

[14] M.S. Lam and R.P. Wilson, "Limits of Control Flow on Parallelism", in *Proc. 19th ISCA*, pp. 46 -57, May 1992.

[15] S.A. Mahlke, D.C. Lin, W.Y. Chen, R.E. Hank and R.A. Bringmann, "Effective Compiler Support for Predicated Execution using the Hyperblock", in *MICRO25*, 1992.

[16] S. Melvin and Y Patt, "Enhancing Instruction Scheduling with a Block-Structured ISA", in *Intern. Journal of Parallel Prog.*, 23(3) pp 221-243, Jun. 1995

[17] M.C. Merten, A.R. Trick, C.N. George, J. Gyllenhaal, and W.W. Hwu, "A Hardware-Driven Profiling Scheme for Identifying Program Hot Spots to Support Runtime Optimization", in *ISCA26*, 1999.

[18] M.C. Merten, A.R. Trick, E. M. Nystrom, R.D. Barnes and W. Mwu, "A Hardware Mechanism for Dynamic Extraction and Relayout of Program Hot Spots", in *ISCA27*, May 2000.

[19] R. Nair and M.E. Hopkins, "Exploiting instruction level parallelism in processors by caching scheduled groups", in *Proc. ISCA24*, pp. 13-25, 1997.

[20] A. Parikh, M. Kandemir, N. Vijaykrishnan and M.J. Irwin, "VLIW Scheduling for Energy and Performance" in *Proc. IEEE Workshop on VLIW*, pp. 111-117. April 2001.

[21] S. Patel, M. Evers and Y. Patt, "Improving Trace Cache Effectiveness with Branch Promotion and Trace Packing", in *ISCA25*, June 1998.

[22] S. Patel and S. Lumetta, "rePlay: A Hardware Framework for Dynamic Optimization", in *IEEE Trans. on Computers*, 50(6), pp 590-608, June 2001

[23] S. Patel, T. Tung, S Bose and M. Crum, "Increasing the Size of Atomic Instruction Blocks using Control Flow Assertions", in *MICRO33*, 2000.

[24] A. Peleg and U. Weiser. "Dynamic Flow Instruction Cache Memory Organized Around Trace Segments Independent of Virtual Address Line", U.S. Patent 5,381,533, Jan. 1995.

[25] M. Postiff, G. Tyson and T. Mudge, "Performance Limits of Trace Caches", in *Journal of ILP,* vol. 1, Oct. 1999.

[26] A. Ramírez, J. L. Larriba-Pey, C. Navarro, J. Torrellas and M. Valero, "Software Trace Cache", in *Proc. ICS13*, pp. 119–126, 1999.

[27] R. Rosner, A. Mendelson and R. Ronen, "Filtering Techniques to Improve Trace-Cache Efficiency", in *PACT'01*, Sept. 2001.

[28] R. Rosner, M. Moffie, Y. Sazeides and R. Ronen, "Selecting Long Atomic Traces for High Coverage", in ICS'03, pp. 2-11, June 2003.

[29] R. Rosner, Y. Almog, M. Moffie, N. Schwartz and A. Mendelson, "PARROT: Power Awareness through Selective Dynamically Optimized Traces", in *PACS'03*, Dec. 2003.

[30] E. Rotenberg, S. Bennett and J. Smith, "A trace cache microarchitecture and evaluation", in *IEEE Trans. on Computers*, 48(2), pp 111–120, Feb. 1999

[31] B. Slechta, D. Crowe, B. Fahs, M. Fertig, G. Muthler, J. Quek, F. Spadini, S. J. Patel and S. S. Lumetta, "Dynamic Optimizations of Micro-Operations", in HPCA9, Feb. 2003.

[32] V. Srinivasan, D. Brooks, M. Gschwind, P. Bose, V. Zyuban, P.N. Strenski and P.G. Emma, "Optimizing Pipelines for Power and Performance", *MICRO 35*, 2002.