Compiler Optimizations for the PA-8000

Anne M. Holler
Hewlett-Packard Company
aholler@cup.hp.com

Abstract

Compiler optimizations play a key role in unlocking the performance of the PA-8000, an innovative dynamically-scheduled machine which is the first implementation of the 64-bit PA 2.0 member of the HP PA-RISC architecture family. This wide superscalar, long out-of-order machine provides significant execution bandwidth and automatically hides latency at runtime; however, despite its ample hardware resources, many of the optimizing transformations which proved effective for the PA-8000 served to augment its ability to exploit the available bandwidth and to hide latency. While legacy codes benefit from the PA-8000's sophisticated hardware, recompilation of old binaries can be vital to realizing the full potential of the PA-8000, given the impact of new compilers in achieving peak performance for this machine.

1 Introduction

The PA-8000 [20,25] is the first implementation of the 64-bit PA-RISC 2.0 architecture [30]. The processor is dynamically scheduled with a 56-entry reorder queue, organized as a 28-entry ALU queue and a 28-entry memory queue; this reorder queue is significantly longer than that of any other microprocessor currently available in the marketplace [34]. The PA-8000 is 4-issue superscalar, with 2 integer arithmetic/logic units, 2 shift/merge units, 2 load/store address units, 2 floating point multiply/accumulate units, and 2 floating point divide/square-root units. Given its lengthy reorder queue and wide issue, the PA-8000 might be expected to subsume certain functionality of the low level compiler optimizer, such as instruction scheduling. However, experience has demonstrated that the combination of innovative hardware and aggressive optimization, including compile-time scheduling, yields the PA-8000's strong performance, which at 180MHz is currently 11.8 peak SPECint95 ratio and 20.2 peak SPECfp95 ratio.

The remainder of this paper is organized as follows. First, the effect of optimization on PA-8000 performance is examined. Next, compiler optimization techniques particularly suited to the PA-8000 are discussed. Performance measurements for the PA-8000 vis a vis competitive systems are then reported. The paper concludes with lessons learned in optimizing for a superscalar out-of-order machine.

2 Performance Impact of Optimization

The performance impact of optimization for the PA-8000 is substantial. Legacy codes and codes not utilizing advanced optimization techniques may not realize the machine's full performance potential.

While the PA-8000 delivers solid speed-up on legacy codes, such codes can suffer significant lost performance opportunity by not being rebuilt for the PA-8000. Peak SPECint95 binaries built for the PA-8000 using the 10.20 version of the PA-RISC compilers run 38% faster on the PA-8000 than do peak SPECint95 binaries built for the PA-7200 using the 10.01 version of the PA-RISC compilers; similarly built peak SPECfp95 binaries run 53% faster. Legacy codes do not exploit new PA-8000 features, may contain PA-8000 hazards, and do not benefit from the latest improvements in the compiler optimizer.

The power of the PA-8000 hardware provides advanced optimization techniques with more leverage to increase performance than was available in past PA-RISC processors. Peak SPECint95 binaries are 46% faster than those built with -O on the PA-8000, while they are only 38% faster than those built with -O on the PA-7200 (all built using the 10.20 compilers). Peak SPECfp95 binaries are 41% faster than those built with -O on the PA-8000, while only 9% faster than -O on the PA-7200.

3 Optimization Techniques for the PA-8000

The optimizations described in this paper (and more!) are incorporated in version 10.20 of the PA-RISC compilers. All HP compilers for the PA-RISC share common optimizing components. The structure of HP compilers and the purpose of each optimizing component are shown in Figure 1 [24]. The user selects desired optimizing components with command-line options; for example, the low level optimizer may be invoked via -O and the high level optimizer is typically invoked via +O3 or +O4. The user may employ the profile-based optimization (PBO) framework to

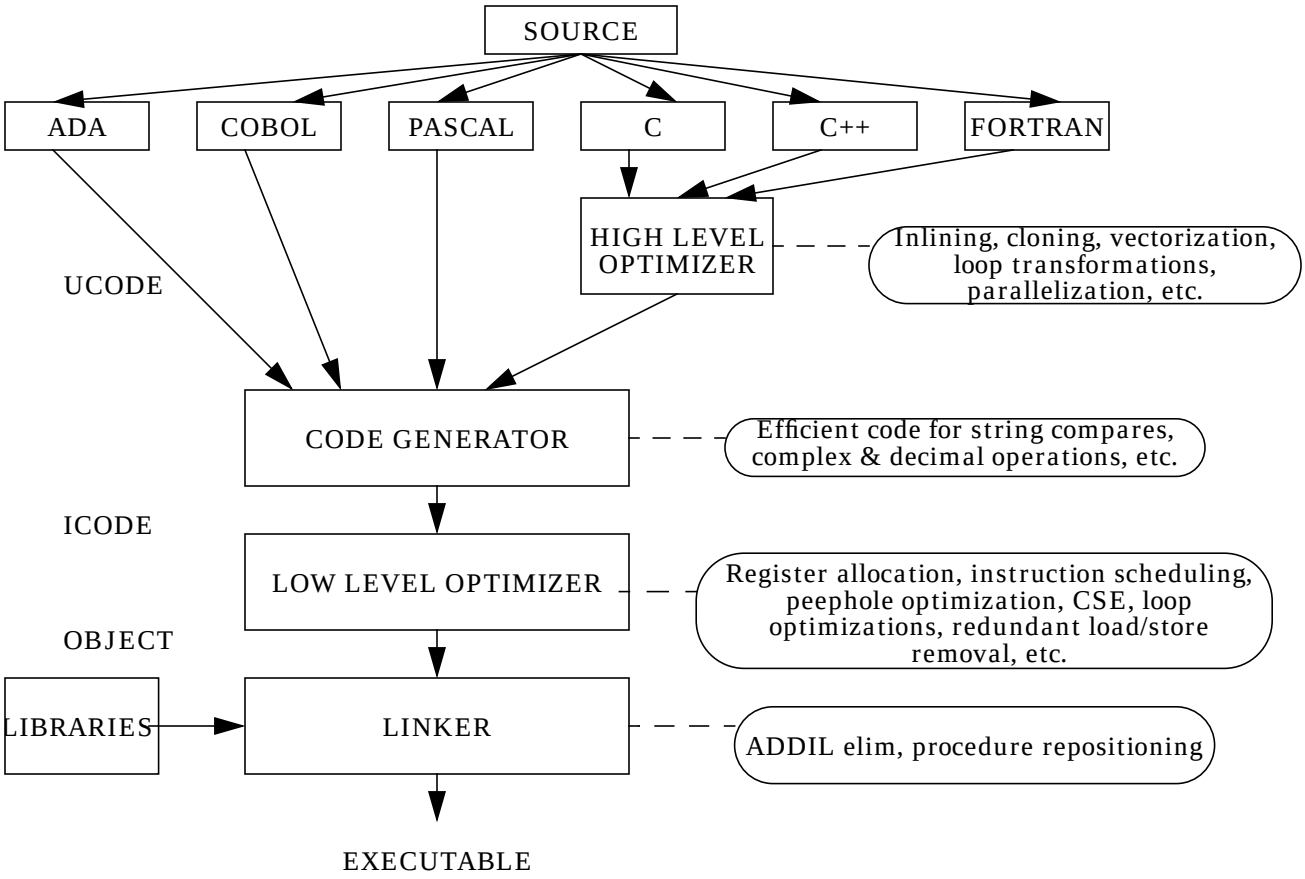

FIGURE 1: Structure of PA-RISC Optimizing Compilers

have data about a program gathered during instrumented runs on representative data sets and fed back into the optimization process.

The PA-RISC Low Level Optimizer (LLO) performs global optimizations within a procedure [29,21]. It consists of the following phases: basic block formation and optimization, interval construction and calculation of reaching dataflow information [27], global common subexpression elimination, calculation of need dataflow information, promotion of memory variables, loop optimizations, formation and optimization of definition/use webs, instruction scheduling, coloring register allocation, peephole optimization, branch optimization, branch reachability fixing, and (again) instruction scheduling. The LLO incorporates many aggressive low level techniques, including register reassociation [40] and software pipelining [38].

The PA-RISC High Level Optimizer (HLO) handles transformations which depend on a high semantic level intermediate representation of the input code and/or on cross-procedure or cross-module processing. Such transformations include inlining, cloning, whole program analysis, parallelization, vectorization, loop interchange, and loop fusion. For the various implementations of the PA 1.0 and PA 1.1 architectures, the HLO was essentially machine-independent; however, for the PA-8000, the HLO's operation was influenced by machine-specific considerations.

Optimization techniques introduced or significantly enhanced in the 10.20 version of the PA-RISC compilers can be divided into four areas: optimization for procedural dependencies, for true dependencies, for resource conflicts, and general and high-level optimization. Procedural dependencies are engendered by branching; on the PA-8000, correctly predicted branches which hit in the branch target address cache incur no penalty, while mispredicted branches may have a relatively large penalty of around 6 cycles. Techniques introduced in this area include loop unrolling, superblock formation, assertion propagation [4,22], if-conversion, static branch prediction hinting [6,37], and switch statement improvements. The latencies associated with producing a value needed by a subsequent operation are true dependencies; the PA-8000's reorder queue is intended to hide these latencies, but its length represents a limitation on its ability to do so. Techniques intro-

duced in this area include data cache prefetching, instruction scheduling for dependent chains [16], strength reduction extensions, store load dependence elimination, and tree height reduction [32]. Resource conflicts reflect hardware constraints and features. Techniques introduced in this area include FDIV/FQRT polarity scheduling [16], nullified delay slot scheduling, prefetch spreading [16,31], single-precision floating point false dependence avoidance, memory access coalescing [13], exploiting PA-8000 instruction set extensions, and two target operation avoidance. Finally, a number of general and high-level optimization techniques were introduced or improved; these include type-inferred aliasing [11], scalar replacement, shrink-wrapping [10], inlining, cache-related transformations, and loop-level transformations. A subset of these techniques are discussed below; more information can be found in [HOLL96].

3.1 Loop Unrolling

Loop unrolling [15], in which a loop body is replicated one or more times with appropriate adjustments to array indices and loop control code, reduces the iteration overhead associated with looping (i.e., increment/decrement, test, and branch). In addition, it exposes more cross-iteration ILP to the optimizer. While the PA-8000 hardware can automatically schedule instructions across iterations, loop unrolling and certain other optimizations it enables were found to be quite effective for this machine. Loop unrolling for the PA-8000 is discussed in detail in [41]; below is a brief description of loop unrolling in the PA-RISC LLO, with PA-8000 issues highlighted.

The fundamental heuristic related to loop unrolling is the selection of the unroll depth; several attributes of the PA-8000 are relevant to the selection criteria. One factor considered is the number of instructions in the loop. For this factor, unroll depth is chosen as a balance between the fact that the PA-8000 can handle substantial code size growth, by virtue of its large (1 megabyte) instruction cache, and the potential side-effects of code size growth, which may include increased register pressure leading to spilling or excessive compile time and space consumption. Another factor considered relates to data cache prefetching. If PA-8000 explicit data cache prefetch instructions (Section 3.4) will be inserted into the loop, and if the memory access stride is less than the cache line length in the original loop, it is desirable to limit the unroll depth so that the stride remains within the cache line length in the unrolled loop, to promote the efficacy of the prefetch overhead reduction algorithm. If recurrent scalar replacement (Section 3.9) has been performed, an unroll depth which matches the recurrence distance allows the removal of copies which carry values across iterations. Critical resource usage is also considered; in particular, the number of long latency floating point divide (FDIV) and square root (FSQRT) operations is counted, because an unroll factor which yields an even number of these operations in the loop body is preferable to aid FDIV/FSQRT polarity scheduling (Section 3.6). The expected trip count (which may be constant, based on profiling information, or derived from static heuristics) is also relevant; choosing an unroll depth which is a multiple of the expected trip count may tend to allow execution of the compensation loop[1] to be avoided. The user may suggest an unroll depth for the program's loops via a command line option. Finally, issues related to managing the unrolled loop are considered in selecting the unroll depth. An unroll depth which is a power of two allows the use of an inexpensive shift operation to determine how many times to execute the compensation loop; in this case, the compensation loop is executed prior to the unrolled loop, so that the index variable associated with the unrolled loop is not live on loop exit, improving the operation of register reassociation for the unrolled loop. An unroll depth which is not a power of two requires a more expensive operation to determine how many times to execute the compensation loop; in this case, the compensation loop is placed after the unrolled loop with its repetition factor determined by the remainder value of the unrolled loop's index variable, which is thus live on exit from the loop impacting register reassociation's efficacy.

Perhaps surprisingly, application code may contain loops which iterate a constant number of times. This occurs, for example, when the code solves a problem which involves a fixed width in certain dimensions. Completely unrolling constant trip count loops and removing all loop overhead, even when the resulting code size growth is nontrivial, proved to be profitable on the PA-8000.

When a loop has a fairly small trip count, its iterate-or-exit branch can have a high misprediction rate. If control passes through this poorly predicted loop fairly often, performance may be impacted. In any such situation which does not lend itself to regular unrolling and for which the small trip count tends to usually have a particular value, the loop body can be replicated that number of times, with each copy retaining its own branch. This simple form of unrolling serves to split the loop's single branch point into several branch points, each of which is more predictable (i.e., each will likely fall through).

3.2 Superblock formation

Superblock formation [26] is an optimizing and an optimization-enabling transformation, which involves duplicating code so that certain incoming edges into a commonly executed path of the control flow graph can be removed. As a result, larger basic blocks are constructed and some control flow overhead is eliminated, reducing pressure on the PA-8000's branch prediction cache (BPC) and thereby improving its prediction accuracy. These changes can in turn be exploited in a number of ways. For example, after constructing superblocks, a common subexpression which was only valid on a certain path through the code could become available on a distinct path (thereby allowing a common subexpression elimination framework to effect a form of partial redundancy elimination). The

1. The compensation loop is executed when there are too few iterations to be handled by the unrolled loop.

PA-8000's tolerance for code size growth and its high branch misprediction penalty make this transformation attractive.

Superblock formation consists of identifying traces [18] which represent frequent paths through the code and then duplicating certain code along those traces, modifying or removing branches as appropriate. The LLO's algorithm is similar to that described in [9]. Using profile information collected via the PBO framework, traces are constructed by repeatedly picking the basic block with the highest execution frequency from a bucket of candidate basic blocks, initially filled with all basic blocks that meet a certain threshold execution level, and then forming an associated trace by selecting adjacent frequently-executed basic blocks. After traces are constructed, decisions concerning whether and how much tail duplication should be done are made. Tail duplication of blocks containing a large percentage of conditional branches is avoided, since introducing multiple copies of those branches can increase conflicts in the PA-8000's BPC. Superblocks are formed during the basic block construction phase, to allow subsequent phases to capitalize on the transformed code. However, some phase ordering issues are associated with this choice and are addressed by not creating superblocks which may interfere with loop optimizations or with if-conversion.

3.3 If-conversion

On the PA-RISC architecture, all computational and branch instructions can nullify the execution of the following instruction. In the case of computational instructions, nullification is performed conditionally, based on the outcome of a test; this conditional nullification represents a simple form of predicated execution [14]. If-conversion is an optimization technique involving the conversion of conditional blocks of code to sequences of conditional simple statements; this technique can serve to transform control flow dependencies into data flow dependencies, by including the condition as an input to the operation [1]. The conditional simple statement semantics of if-conversion can be realized using nullification on a computational instruction, thereby avoiding branching. Surprisingly, although avoiding branching is quite desirable on the PA-8000, developing heuristics for deciding when to perform if-conversion was difficult.

The problem is that there are certain overheads associated with nullification on the PA-8000. Nullification engenders a latency; a non-branch conditionally nullified operation is not considered data-ready until its potentially nullifying instruction has launched for execution. Note that there would be no such latency if the instruction were simply located on a correctly-predicted branch path. More importantly, when an instruction is nullified, the dependencies which were established in the reorder queue are affected. In this regard, nullification is an anomaly for an out-of-order machine. The PA-8000 hardware accelerates the nullify-slot operation of certain instructions; these instructions are said to be in *select* convention.

A detailed description of if-conversion for the PA-8000 can be found in [3]. Factors governing its application include the amount of conditional code, the balance between the amount of then and else code for if-then-else constructs, and the execution frequency on alternative paths through the control flow graph. Producing candidate if-converted code sequences involves careful selection and efficient utilization of guarding conditionals, the use of unconditional code when appropriate, and the exploitation of select convention instructions. Estimated best-case processor cycles for the original and if-converted code are compared; computing these cycle counts was complicated by the difficulty in calculating cycles for an out-of-order machine, particularly with respect to expected misprediction costs [3]. If-conversion, judiciously applied, proved very effective for the PA-8000.

3.4 Data Cache Prefetching

A cache miss on the PA-8000 has a latency which may be in excess of 50 cycles. To aid in hiding this latency, the PA-8000 supports the explicit data cache prefetch instructions defined by the PA 2.0 architecture. Encoded as integer loads to general register zero, these instructions (unlike normal loads) do not engender exceptions or traps and are not subject to retirement delays pending service from the cache.

Compiler-generated data cache prefetch [35] for the PA-8000 is discussed in detail in [19]; a brief overview is given here. The LLO will insert explicit data prefetch instructions into loops (including those with internal control flow) which contain regular data access patterns in terms of effective memory strides. Worst-case cache alignment of user data is taken into account, and runtime dimensioned arrays are supported. Given the expected cache miss latency, the number of loop iterations in advance of which to prefetch data is determined by the expected loop iteration latency, which depends on the composition of the loop and on PA-8000 scheduling information. The prefetch algorithm focuses on innermost loops, but prefetches for non-innermost loops as well, if they are expected to be executed relatively more often each time they are entered than loops nested within them. One interesting situation occurs when an enclosing loop is expected to be executed more often each time it is entered than a nested loop, and the two loops have conflicting stride directions. In this case, the LLO will reverse the direction of the prefetches in the nested loop, so that it will effectively be prefetching for future iterations engendered by the outer loop.

Data prefetch can increase performance on the PA-8000 dramatically. For several benchmarks, the speedups were between 50 and 100%. Unfortunately, data prefetch can also diminish performance. Prefetch instructions are added selectively to minimize their overhead without unduly compromising miss coverage by exploiting apparent group spatial and group temporal locality among references to different elements of the same array. Nevertheless, explicit data prefetch instructions consume retirement bandwidth and increase cache accesses, and these costs may not be offset by a corresponding decrease in data cache miss over-

head, either because the prefetched data is already cache-resident (and in general, the compiler cannot determine this statically), or because the prefetched data displaces (or is displaced by) other needed cache data, or because the prefetched data is not actually used by the application. Due to the risk of performance loss, data prefetch is only performed at the user's request.

3.5 Strength Reduction

On the PA-8000, FDIV is a relatively long latency operation (17 cycles single precision, 31 cycles double precision). Hence, when a division is to be performed multiple times with the same divisor, it is preferable to compute the reciprocal and then to replace the division operations with multiplications. This strength reduction technique, termed FDIV weakening, is performed by the LLO in a variety of situations, involving both loops and straight-line code. Note that the technique can affect floating point accuracy, and is only applied when the user has indicated that this type of optimization is acceptable.

The PA-RISC instruction set architecture does not include multiply or divide operations on operands in the integer registers. Several strength-reducing transformations are related to this design decision. Integer multiplication by a constant is replaced by a series of shift-add instructions. Integer division and remainder after division by a constant are computed using a floating point multiplication by the reciprocal, which is produced as a literal at compile time. Integer division and remainder after division by a variable are calculated using a floating point multiplication by the reciprocal, which is produced at run time.

3.6 FDIV/FSQRT Polarity Scheduling

The PA-8000 has two FDIV/FSQRT units. However, to simplify the hardware, there is a constraint on both units being used simultaneously, which is that the two associated instructions must be of opposite polarity, i.e., one must have an even and the other an odd slot address in the reorder queue. Because the units are not pipelined and have a long latency, a substantial performance benefit can be derived in some cases from laying out the code so that FDIVs and FSQRTs which are executed in close temporal proximity have opposite polarity.

The final instruction scheduling phase performs polarity scheduling in several situations. For a single basic block containing more than one FDIV or FSQRT, it tracks ALU queue slot addresses and positions code so that successive independent FDIVs or FSQRTs are of opposite polarity. For a single basic block loop containing an FDIV or FSQRT instruction without loop-carried dependence, it ensures that the loop has odd number of ALU instructions, by padding with a NOP if necessary, so that the same instruction will flip polarity on each successive iteration. If-conversion, loop unrolling, superblock formation, and assertion propagation can aid polarity scheduling by transforming several basic blocks into a single basic block.

3.7 Single-precision Floating Point False Dependence Avoidance

On the PA-RISC 1.1 and later architectures, there are 32 64-bit (double precision) floating point registers, which can also be used as 64 32-bit (single precision) floating point registers. To save hardware, the PA-8000 computes register dependencies with respect to double precision registers. Hence, if frX and frY are single precision registers which form the two halves of a double precision register, a false data dependence is recorded when frX is used within the same reorder window in which frY is defined. Because of this (false) dependence, the use instruction is not launched until the definition instruction completes execution. When the use instruction is launched, the fact that its established dependency is incorrect is detected before it can retire and the instruction is aborted, with its queue slot being marked for subsequent relaunching. Dependent instructions may be launched and aborted as well. Instructions which record false dependencies are aborted and relaunched approximately every 4 cycles until the source of the false dependence is retired, wasting bandwidth.

To address this problem, the coloring register allocator was revised to not assign both halves of a double precision register to two single precision items with overlapping lifetimes. While this "wasting" of registers might be expected to impact performance, the benefit of avoiding single precision false dependence outweighed any penalty from increased register pressure. The final instruction scheduling phase was also modified to not reorder instructions in a way which would introduce single precision false dependence.

3.8 Exploiting PA-8000 Instruction Set Extensions

PA 2.0 is the first 64-bit member of the PA-RISC family. Until the introduction of PA 2.0, 64-bit data items consumed two registers and operations on those items was handled via millicode calls. On PA 2.0, 64-bit data items can be kept in a single register and operations on those items can be performed inline.

PA-RISC integer load and store operations support 14-bit immediate offsets, but prior to PA 2.0, floating point load and store operations only provided 5-bit immediate offsets. The PA 2.0 instruction set extended the offsets on floating point memory operations to match those on integer memory operations. Use of the wider offsets results in tighter code sequences and, in particular, in improved register reassociation.

The shift amount register (sar) is a 6-bit control register which is used by the variable versions of the shift, extract, deposit, and branch-on-bit instructions. For members of the PA-RISC family introduced before PA 2.0, the following code sequence was produced for 32-bit signed integer "z = x << y":

```
   SUBI        -1,y,temp   !1s cmplt for DEPW bitpos
   MTSAR       temp        !Move temp to sar
   DEPW,Z      x,sar,32,z  !Bits #d most to least signf
```

and for "z = x >> y":

```
   SUBI        -1,y,temp   !1s cmpl for EXTRW bitpos
   MTSAR       temp        !Move temp to sar
```

```
EXTRW,S     x,sar,32,z   !Bits #d most to least signft
```

The PA 2.0 ISA introduced MTSARCM (move ones complement to sar), which allows shorter idioms; for 32-bit signed integer "z = x << y", the PA 2.0 code is:

```
MTSARCM  y
DEPW,Z       x,sar,32,z
```

and for "z = x >> y", the PA 2.0 code is:

```
MTSARCM  y
EXTRW,S     x,sar,32,z
```

PA-RISC floating point comparisons set a compare bit (C-bit) in the floating point status register. Prior to the advent of PA 2.0, there was a single C-bit; PA 2.0 introduced multiple C-bits. The instruction scheduler utilizes multiple C-bits to hide latency, in sequences involving several independent code streams containing if-conversion.

FMPYFADD and FMPYNFADD are PA 2.0 instructions which implement floating point multiply and accumulate operations. These instructions take 4 operands, rb, ra, rd, and rc; FMPYFADD produces rc = rd + (rb * ra) and FMPYNFADD produces rc = rd - (rb * ra). On the PA-8000, each of the floating point instructions FMPYFADD, FMPYNFADD, FMPY, FADD, and FSUB takes the same number of cycles, so it is profitable to combine an FMPY-FADD or an FMPY-FSUB sequence into a single FMPY-FADD or FMPYNFADD instruction. These instructions are synthesized for the PA-8000 target during the first instruction scheduling phase,[1] as explained in [5]. Several situations are considered. The most straightforward case is when the result of an FMPY is used only once, by an FADD or as the subtrahend of an FSUB; this substitution is essentially a peephole transformation. More complicated is the case when both operands of an FADD are distinct FMPYs; this involves a choice as to which of the two FMPYs should be included in the substitution. The PA-8000 instruction scheduler chooses to reduce the number of instructions on the code's critical path by selecting the FMPY with the longer path from the root nodes (the nodes near the start of the basic block). Another case is when a single FMPY feeds into two FADDs. Two FMPYFADDs are generated, unless the associated extension of the original FMPY's operands' live ranges engenders too much register pressure, in which case no transformation is performed. Finally, when a single FMPY feeds an FADD or FSUB eligible for synthesis, but there are other consumers of the FMPY result, the FMPY is duplicated and the FMPYFADD or FMPYNFADD is generated as appropriate; this shortens the path length to the result of the FMPY-FADD or FMPYNFADD, even though the total number of operations remains the same.

1. The generation of similar instructions (FMAs) is handled by the front-end of IBM's RS/6000 compiler, rather than by its optimizer. This approach allows FMAs to be produced even when optimization is not enabled, but may have difficulty recognizing certain situations in which FMAs can be employed and making trade-offs with respect to critical path when there is a choice concerning where to introduce an FMA.

3.9 Scalar Replacement

Scalar replacement is a technique in which subscripted variables that are reused in innermost loops are assigned to scalar temporaries to facilitate their allocation to registers [7]. The PA-RISC LLO also applies the technique to loops with control flow [8] and to non-innermost loops. In addition, the LLO scalar-replaces non-subscripted variables, including structure field references and references through pointers, and uses the technique to provide a general framework for loop-duration promotion of scalar variables whose complete lifetimes do not warrant register promotion [17]. Conditional references are scalar-replaced, if the relevant address expression can be proven safe in its unconditional setting, subject to profitability. Scalar replacement is also applied to recurrent memory references, after which predictive commoning [36] is used to remove recurrent common subexpressions and to optionally perform recurrent FDIV weakening. Loop unrolling allows the associated copies in the loop body to be removed.

While performing scalar replacement, the LLO register-promotes the scalar temporary; register pressure heuristics are used to avoid over-allocation. Several transformations are applied in association with the register promotion. If an item with a loop invariant address is scalar-replaced and its reuse set within the loop begins with an unconditional store, then the scalar load which would usually be placed in the loop header is not materialized; similarly, if an item with a loop variant address is scalar-replaced and its reuse set within the loop begins with an unconditional store, then all associated loads located in the loop body are deleted. A loop variant reuse set is usually not scalar-replaced if the first load or last store is conditional; however, a reuse set containing a conditional first load is transformed if preceded by an unconditional store, and a reuse set containing a conditional last store is transformed if an unconditional store can safely and profitably be inserted at the end of the loop.

Note that scalar replacement can have beneficial effects on the PA-8000 which are somewhat indirectly associated with register promotion. If a load is launched out-of-order before a store to the same address then, when the store retires, entries in the reorder queue are purged and instructions following the store are refetched. (This scenario is only likely when the store is located in a potentially nullified slot and the following load is not.) Since the LLO's scalar replacement can register-promote in this situation, this expensive sequence is averted.

3.10 Inlining

Inlining, in which a subprogram call is replaced by a copy of the callee's body with suitable substitutions for parameters and conflicting identifiers [12], improves performance on models across the PA-RISC family. However, it is particularly effective on the PA-8000, and heuristics governing its application were tuned accordingly. Not only does the PA-8000 have a large instruction cache, but also the subroutine return branch, which is indirect through a

register, is not predicted on the PA-8000. Calls through pointers, which are also register indirect and not predicted, are potential bottlenecks as well. The HLO uses a combination of caller/callee site matching and profiling information to introduce a runtime test against the address of an indirect call's likely target, to choose between making the call and executing an inlined version of the recognized target. In addition, aggressive inlining yields benefits which compare favorably vis a vis interprocedural analysis in terms of facilitating other optimizations [39]; in some cases, significant speed ups were realized on the PA-8000 by flattening entire call trees.

3.11 Cache-Related Transformations

The HLO performs two categories of cache-related transformations: those intended to enhance locality and those intended to reduce various kinds of conflicts. Using the loop transformation framework described in [33], the HLO enhances locality by interchanging loops to place stride one accesses in the innermost loop. Loop fusion sometimes serves to improve locality as well. The PA-8000 uses a direct-mapped caching scheme, and cache conflicts between data items used in close temporal proximity occur in some codes. The HLO, under a user option, will attempt to reduce cache conflicts by inserting padding between items stored adjacently, so as to modify their addresses appropriately. In addition, the PA-8000's data cache is non-blocking; miss requests for up to ten distinct cache lines can be outstanding at a given time and cache misses which map to different banks are handled with less latency than those which map to the same bank. The HLO can also insert padding to reduce memory bank conflicts.

3.12 Latency-Related Loop Optimizations

Several HLO loop optimizations which consider latency were particularly important for PA-8000 performance. Vectorization of reduction loops increased the ILP available for exploitation by the PA-8000. A reduction is an operation that computes a scalar value from an array; a reduction loop is serialized with respect to accesses to that scalar value. However, if the operation is associative, it can be vectorized by computing separate scalar values for subpartitions of the array & then combining those values appropriately to compute the final scalar value [2]. HLO recognizes certain reduction loops and replaces them with calls to vectorized library routines.

Another key latency-related HLO loop transformation was loop interchange to make long latency operations loop invariant. For example, in the following code, the divisor varies in the inner loop.

```
      DO 10 J=1,N
            DO 10 I=1,M
10                X(I,J) = X(I,J) - (Z/Y(I))
```

However, if the loop were interchanged, then the divisor would become inner loop invariant, and the division could be moved out of the inner loop, reducing the number of long latency divisions done overall. Note that this motivation for loop interchange may conflict with that of enhancing spatial locality; in this example, interchanging will move stride one accesses out of the inner loop.

4 Competitive Performance

The authors chose to report performance with respect to the SPEC95 benchmarks, since these codes are widely available and familiar to many. The PA-8000 optimization results for SPEC95base are given in the table below, along with key competitors' results [34]. Optimization contributes strongly to the PA-8000's results; legacy binaries built for the PA-7200 achieve 7.8 SPECint95 and 9.0 SPECfp95 when run on the PA-8000.

SPEC95BASE	INT	FP
HP PA-8000 180MHz	10.8	18.3
Digital 21164 500MHz	12.6	18.3
SGI R10000 200MHz	8.9	17.2
Sun Ultrasparc 200MHz	8.5	15.0

This paper has focused on the effect of compiler optimizations on performance, but many other factors contributed directly to the PA-8000's SPEC95 results, including the chip and system designs, the runtime library, the front-ends, the intermediate code generator, the linker, and the operating system. Indirect contributions are too numerous to enumerate, but one that deserves special mention is that of the performance simulator. The complexity of this machine made compile-time estimations of performance obsolete, and made accurate simulation crucial to determining the efficacy of optimizing transformations. Simulation measurements exposed critical tuning opportunities and brought to light counterintuitive attributes of code behavior. In addition, once prototype hardware was available, the on-chip performance monitoring functionality was very useful as well.

5 Conclusion

The synergy between innovative hardware and aggressive optimization yields PA-8000's industry-leading performance. As has been suggested is the case for superscalar out-of-order machines [28], optimizing for procedural dependencies had a substantial impact on performance. Developing appropriate heuristics for such optimizations was challenging. Optimizing for true dependencies was more important than expected for this machine, given that it has a larger reorder queue than any other microprocessor in the marketplace. Optimizing for resource conflicts was important as well, despite the fact that the machine has a generous number of processing units and registers. High level optimization was influenced by machine-specific considerations.

6 References

[1] Allen, J.R., Kennedy, K., Porterfield, C., and Warren, J., "Conversion of Control Dependence to Data Dependence", *POPL*, January 1983.

[2] Bacon, D., Graham, S., and Sharp, O., "Compiler Transforma-

tions for High-Performance Computing", *ACM Computing Surveys*, Vol. 26, No. 4, December 1994.
[3] Benitez, Manuel, "Exploiting Conditional Nullification on the PA-8000", *HP Technical Report*, 1996.
[4] Blume, W. and Eigenmann, R., "Symbolic Range Propagation", *Proceedings of the 9th International Parallel Processing Symposium*, April 1995.
[5] Blume, W., "Effective Floating Point Multiply Fused Add Code Generation", *HP Technical Report*, 1996.
[6] Burch, Carl, "PA-8000: A Case Study of Static and Dynamic Branch Prediction", *HP Technical Report*, 1996.
[7] Callahan, D. Carr, S., and Kennedy, K., "Improving Register Allocation for Subscripted Variables", *PLDI*, June 1990.
[8] Carr, S. and Kennedy, K., "Scalar Replacement in the Presence of Conditional Control Flow", *Software - Practice and Experience*, January 1994.
[9] Chang, P.P., Mahlke, S.A., and Hwu, W.-M., "Using Profile Information to Assist Classic Code Optimizations", *Software - Practice and Experience*, Vol. 21, December 1991.
[10] Chow, F.C., "Minimizing Register Usage Penalty at Procedure Calls", *PLDI*, June 1988.
[11] Coutant, D.S., "Retargetable High-Level Alias Analysis", *POPL*, January 1986.
[12] Davidson, J. and Holler, A., "Subprogram Inlining: A Study of its Effects on Program Execution Time", *IEEE Transactions on Software Engineering*, Vol. 18, No. 2, 1992.
[13] Davidson, J. and Jinturkar, S., "Memory Access Coalescing: A Technique for Eliminating Redundant Memory Accesses", *PLDI*, June 1994.
[14] Dehnert, J.C., Hsu, P.Y.-T., Bratt, J.P., "Overlapped Loop Support on the Cydra 5", *ASPLOS*, April 1989.
[15] Dongarra, J.J., and Hinds, A.R., "Unrolling Loops in FORTRAN", *Software - Practice and Experience*, Vol. 9, No. 3, 1979.
[16] Dunn, D.A. and Hsu, W.-C., "Instruction Scheduling for the HP PA-8000", *MICRO-29*, December 1996.
[17] Ebcioglu, K., Groves, R.D., Kim, K., Silberman, G.M., and Ziv, I., "VLIW Compilation Techniques in a Superscalar Environment", *PLDI*, June 1994.
[18] Fisher, J.A., "Trace Scheduling: A Technique for Global Microcode Compaction", *IEEE Transactions on Computers*, C-30, 7, July 1981.
[19] Gornish, E., "Data Prefetching on the HP PA-8000", *HP Technical Report*, 1996.
[20] Gwennap, Linley, "PA-8000 Combines Complexity and Speed", *Microprocessor Report*, November 14, 1994.
[21] Hanson, R., "New Optimizations for PA-RISC Compilers", *Hewlett-Packard Journal*, June 1992.
[22] Harrison, W., "Compiler Analysis of the Value Ranges for Variables", *IEEE Transactions on Software Engineering*, SE-3(3), May 1977.
[23] Holler, A.M., "Optimization for a Superscalar Out-of-Order Machine", *MICRO-29*, December 1996.
[24] *HP PA-RISC Compiler Optimization Technology White Paper*, HP Part No. 5963-7250E, March 1995.
[25] Hunt, Doug, "Advanced Performance Features of the 64-bit PA-8000", *COMPCON 1995 Digest of Papers*, March 1995.
[26] Hwu, W.W., et. al., "The Superblock: an Effective Technique for VLIW and Superscalar Compilation", *The Journal of Supercomputing*, 7(1,2), 1993.
[27] Jain, Suneel and Thompson, Carol, "An Efficient Approach to Data Flow Analysis in a Multiple Pass Global Optimizer", *PLDI*, June 1988.
[28] Johnson, M., *Superscalar Microprocessor Design*, Prentice-Hall, Englewood Cliffs, NJ, 1991.
[29] Johnson, M.S. and Miller, T.C., "Effectiveness of a Machine-Level, Global Optimizer", *Symposium on Compiler Construction*, June 1986.
[30] Kane, Gerry, *PA-RISC 2.0 Architecture*, Prentice-Hall, Englewood Cliffs, NJ, 1996.
[31] Kerns, D.R. and Eggers, S.J., "Balanced Scheduling: Instruction Scheduling when Memory Latency is Uncertain", *PLDI*, June 1993.
[32] Kuck, D.J., *The Structure of Computers and Computations*, Wiley, New York, 1978.
[33] Li, W., "Compiling for NUMA Parallel Machines", *Ph.D. Thesis*, Cornell, 1993.
[34] "Chart Watch: RISC Processors", *Microprocessor Report*, October 7, 1996.
[35] Mowry, T.C., Lam, M.S., and Gupta, A., "Design and Evaluation of a Compiler Algorithm for Prefetching", *ASPLOS*, 1992.
[36] O'Brien, K., Hay, B., Minish, J., Schaffer, H., Schloss, B., Shepherd, A. Zaleski, M. "Advanced Compiler Technology for the RISC System/6000 Architecture", IBM, 1991.
[37] Pettis, Karl and Hansen, Robert, "Profile Guided Code Positioning", *PLDI*, June 1990.
[38] Ramakrishnan, Sridhar, "Software Pipelining in PA-RISC Compilers", *Hewlett-Packard Journal*, June 1992.
[39] Richardson, S. and Ganapathi, M., "Interprocedural Analysis vs. Procedure Integration", *Information Processing Letters*, Vol. 32, 1989.
[40] Santhanam, Vatsa, "Register Reassociation in PA-RISC Compilers", *Hewlett-Packard Journal*, June 1992.
[41] Shah, Lacky, "Intelligent Loop Unrolling", *HP Technical Report*, 1996.