To appear at
The 34th Annual International Symposium on Microarchitecture, December 2001.

Enhancing loop buffering of media and telecommunications applications using low-overhead predication

John W. Sias Hillery C. Hunter Wen-mei W. Hwu
Center for Reliable and High-Performance Computing
Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign
{sias, hhunter, hwu}@crhc.uiuc.edu

Abstract

Media- and telecommunications-focused processors, increasingly designed as deeply pipelined, statically-scheduled VLIWs, rely on loop buffers for low-overhead execution of simple loops. Key loops containing control flow pose a substantial problem—full predication has a high encoding overhead, and partial predication techniques do not support if-conversion, the transformation of general acyclic control flow into predicated blocks. Using a set of significant media processing benchmarks, drawn from MediaBench and contemporary telecommunications standards, we explore a compromise approach. We demonstrate a compiler using if-conversion and specialized loop transformations to arrange for 70-99% of fetched operations to come from a simple, statically managed 256-instruction loop buffer, saving instruction fetch power and eliminating branch penalties. To complement this we introduce a "niche" form of predication specialized to permit general if-conversion with only a single bit in the encoding of each operation and to eliminate much of the hardware overhead of a predicate register-based approach.

1. Introduction

Most modern high-performance DSP and media processor implementations (*e.g.*, the TI 'C6x, Lucent DSP16000, TriMedia, and StarCore 140) are based on a VLIW design paradigm, with good reason: VLIW offers wide issue (today up to eight operations per cycle) with relatively little instruction issue overhead, clustering is natural and offers enhanced scalability [1], and compiler techniques such as software pipelining [2] effectively employ the VLIW's many processing units in a wide variety of loop kernels.

In the embedded market, where power margins dictate use of the lowest possible clock frequency to achieve a given processing rate, cycles cannot be wasted waiting for branch resolution and instruction fetch. Traditional solutions such as branch predictors and instruction caches are usually considered too costly and undependable for inclusion in such processors. Thus, the statically-scheduled processor has difficulty dealing efficiently with control flow. Both elements are often "replaced" with a dedicated loop buffer—effectively a software-controlled, straight-line-code cache with knowledge of counted loops—allowing efficient looping fetch. Buffering offers benefits including accurate loop-back branch prediction, reduced power consumption due to localization of fetch, elimination of taken-branch bubbles, and, if data and instruction fetch share the same memory bus, reduced bus contention in key loop kernels [3]. Providing a useful buffer model and utilizing it effectively are thus important design and compilation goals.

A variety of techniques [4, 5] have addressed loops containing internal control flow, a special problem for VLIWs, but embedded implementation constraints discount several options. In general, loop buffers accommodate only simple loops—straight-line blocks of code with a loop-back branch at the end—or in some restricted cases, loop nests. In general-purpose, VLIW-styled architectures such as Intel's Itanium, on the other hand, full predicated execution support (a predicate register file, predicate defining operations, and a guard predicate operand on each operation) allows the compiler to implement general control without branches [6]. Such an approach, however, is considered impractical in embedded processors because of the encoding cost of the guard operands (operations on Itanium are 41 bits in length). Media and telecommunications processors therefore typically incorporate a form of *partial* predication, the addition of a conditional move operation or a few simple condition codes capable of guarding the execution of a handful of opcodes. As these approaches do not support *if-conversion* [7], the traditional, general algorithm for generating predicated code, compilers have difficulty taking best advantage of these operations. Hand assembly coding or tuning is today often required to take advantage of the buffers.

In this paper, we combine specialized compiler methods and a new predication implementation to address buffering of loops containing control flow. Our compiler transforms a set of media processing benchmarks (not kernels) such that 89.0% of operations fetched come from a 256-operation loop buffer (compared to 38.7% without the transforma-

tions), while achieving a 37.6% reduction in execution cycles.[1] To complement these techniques, we present a model of predication that is generally useful for execution of loop kernels, that consumes only a single bit in each operation, and that does not significantly increase the complexity of bypass logic. We propose that this form of predication is as applicable, within the benchmark set and architectural assumptions, as full predication, with its predicate register file and guard operands. In these demonstrations we assume a simple, addressable-memory style loop buffer which can be implemented in a variety of microarchitectural styles. In the end an interesting paradox emerges: carefully applying code *expanding* transformations with the goal of increasing fetch regularity allows improved utilization of simple loop buffering, significantly reducing instruction fetch overhead and power consumption.

2. Architecture and application background

Development of image and signal processing in third-generation (3G) cellular and other hand-held products has brought about significant changes in the design of DSP and embedded processors. Along with a shift from a Harvard memory architecture, with dedicated data and instruction memories, to a more general-purpose von Neumann architecture, specially-tailored complex instruction sets are being replaced with deeply pipelined, statically-scheduled LIW and VLIW designs, allowing greater throughput and generality with lower hardware overhead. Given the impact of proportionally long (generally 3 to 5 cycle [8]) branch penalties on the performance of tight DSP loops, architects have also begun to include forms of conditional execution and loop support into their instruction sets. Consensus on the best implementation has not yet been reached.

Texas Instruments' 'C6x line, one of the first DSP families to adopt a VLIW style and a 32-bit datapath, architects eight issue slots and provides for operation execution to be guarded by condition codes [8]. Four bits of each opcode indicate which of five condition registers guards the instruction and whether a zero or a non-zero value causes the instruction to be nullified. The 'C6x line exposes five branch delay slots rather than incorporating a special mechanism for *zero-overhead looping*, the ability to execute counted loops without incurring a loop-back branch penalty

The StarCore SC140 processor includes both a loop buffer and predication support in a 16-bit instruction set. Four sets of loop registers allow for four levels of nested counted loop execution. Using this feature, however, does not guarantee elimination of branch-back overhead, and requires that a host of restrictions be observed [3]. Such a feature is likely useful in hand-coded kernels. SC140 has a single condition register which can be used to guard the odd, the even, none, or all of the operations in each bundle.

[1] excluding *mpeg2enc* and *jpegenc* from *MediaBench*.

Table 1: Benchmarks

Benchmark	Description and input
adpcm{*enc*|*dec*}	ADPCM codec. Input: clinton.pcm
g724{*enc*|*dec*}	ETSI GSM 06.60 speech transcoding [10] Input: 363 frames, speech and ambient noise
jpeg{*enc*|*dec*}	(Mediabench) Independent JPEG Group photo codec. Input: testimg.jpg
mpeg2{*enc*|*dec*}	(Mediabench) Video codec. Input: mei16v2.m2v
mpg123	MPEG-2 Layer 3 audio decoder. Input: short.mp3
pgp{*enc*|*dec*}	Pretty Good Privacy codec. Input: pgptest.plain

One of the most flexible hardware loop support systems commercially available is found on the ST120 DSP/Microcontroller core, a scoreboarded 4-issue LIW with 32-bit instructions [9]. The ST120 provides hardware support for up to three loops, which may be nested, overlapped, or independent of one another. Instruction selection is limited within the loop bodies. Like the 'C6x, most opcodes can be guarded by condition codes; some instructions, however, have access to only one condition register, while others may choose from among up to 16.

Many real systems today incorporate both one of the above DSP cores and a controller or host processor. This fact, combined with the proprietary nature of many commercial algorithmic implementations, limits the set of complete media and telecommunications applications available for study in C source code. Thus, we are currently limited to *MediaBench* [11], which focuses on media and communications tasks, and other commonly available "component" applications, still a significant advance over kernel-only studies. Table 1 lists the benchmarks used in this study. We replaced MediaBench's *g721* benchmark with a more up-to-date and more complex codec, which we call *g724* [10]. The goal of this set of benchmarks is to represent the algorithms supported by near-future hand-held wireless devices, or by base stations accommodating communications with such devices, both of which are highly power- and performance-sensitive applications. Both *jpegenc* and *mpeg2enc* push this envelope, and are included only for comparison. The benchmarks were compiled and simulated in the IMPACT Research Environment, in which intrinsic emulation support implements operations such as saturating arithmetic which would be provided on a DSP-oriented processor.

3. Compilation techniques

In a VLIW context, the opposition of branches to parallelism must be overcome by the compiler. The techniques we describe are not "optimizations" in the traditional sense, as they can and do result in the execution of more *operations* rather than fewer. On the contrary, however, they improve overall execution efficiency by using otherwise idle resources to mitigate branch penalties, such as misprediction latency, hard-to-fill delay slots, and control-based

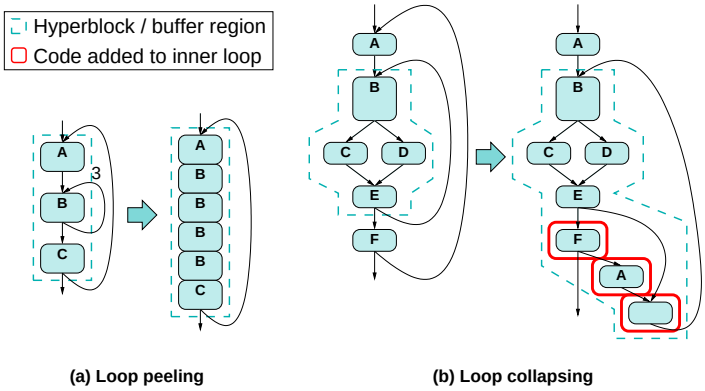


Figure 1: Nested loop transformations

dependence height. This places the following transformations, implemented in the context of the IMPACT compiler framework [12], in the general realm of instruction-level parallel (ILP) transformations.

As previously stated, the use of a loop buffering technique requires removal of all internal control flow from the loop to be buffered. If-conversion [7] converts any *acyclic* region of control flow into an equivalent single-entry, straight-line segment of code, called a *hyperblock* [13]. If the entire body of a loop can be if-converted, it can then be buffered (provided the buffer is of sufficient size). This alone suffices for loops with acyclic internal control flow.

For nested loops, however, other techniques must be applied. Buffering the outer loop, as opposed to just the inner loop, is beneficial when the instruction and trip counts of the inner loop are small—a situation that abounds in media and telecommunications applications. Rather than proposing a more complicated loop buffer, which could handle some limited cases of multiply-nested loops, we present two compiler techniques that transform nested loops into simple ones. Figure 1 illustrates these techniques. In (a), the loop with header "A" contains an inner loop which is known to have four iterations. Provided that the inner loop contains a reasonable number of instructions, it can be eliminated by peeling it completely. We heuristically peel any counted loop of less than six iterations, so long as peeling would create less than 36 instructions.

When, however, block "B" contains a substantial amount of code or is of unknown duration, loop peeling is less attractive because of the code expansion cost. When the number of instructions in the outer loop is small relative to the inner loop, and when the number of iterations of the inner loop in any given iteration of the outer loop is not excessive, there exists another alternative. Figure 1(b) shows the effect of *predicated loop collapsing*, which pulls code from an outer iteration into an inner iteration body. Loop collapsing is a documented technique for converting a nested loop iterating over a matrix into single loops operating linearly (in this case the outer loop is trivially reducible). The form presented here allows *any* outer loop to be pulled in by predicating the outer loop code so that it executes no more frequently than it originally did. If the impact of absorbing the instructions of the outer loop (those in blocks "A" and "F" in (b)) is smaller than the overhead of entering and leaving the loop buffer and taking the outer loop-back branch, this results in increased performance and increased loop buffer efficiency. Given typically long branch resolution latencies and the fact that there are almost always at least a few NOPs in even the optimal modulo schedule of the inner loop, predicated loop collapsing is often beneficial.

Care must be exercised in choosing loops for collapsing, however, because incorporating instructions from outer loops into the inner loop can engender both resource and dependence penalties. Figure 2 shows a doubly-nested loop from the *MediaBench* benchmark *mpeg2dec*. The source code, shown in (a), and the loop's initial machine-level representation, shown in (b), indicate a low trip count inner loop with a small amount of code in its parent loop. (Note that (b) shows the loop after traditional loop optimizations have been applied, including the promotion of `*bp` to a register for the duration of the loop.) Although small enough to have been peeled, this instance also provides a convenient example of the application and benefit of collapsing. Here, as in general, collapsing avoids the static code expansion associated with peeling. In collapsing and if-converting this loop, the first and third blocks shown in (b) are pulled into the inner loop body and guarded under a predicate that causes them to execute only on each eighth iteration of the resulting simple loop, resulting in the bufferable form shown in Figure 2(c). This is accomplished as shown in Figure 1(b), by replicating these blocks into a new hammock (as was done with blocks "A" and "F" in the previous example) prior to if-conversion. Finally, (d) shows the code after further transformations and installation of a special counted loop branch set to 64 iterations, the total number of iterations of the inner loop. (In (c) and (d), the new loop preheader, corresponding to the first block in (b),

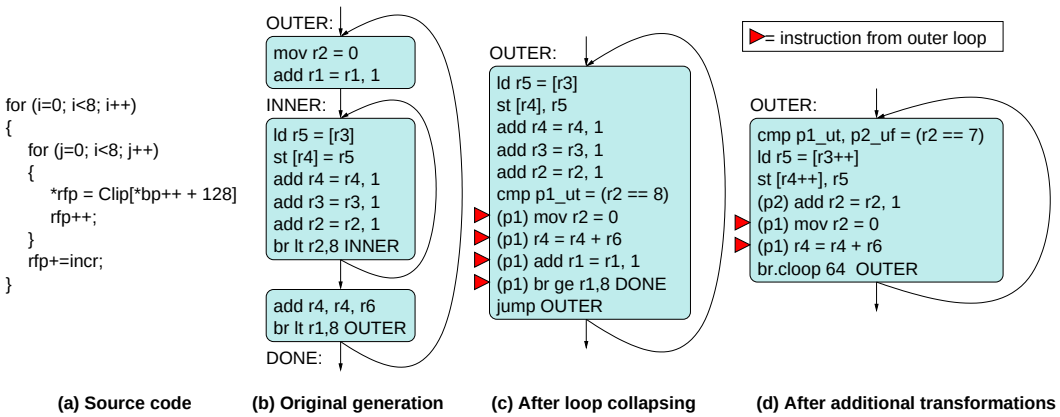


Figure 2: Loop collapsing code example from *mpeg2dec* Add_Block()

is not shown.)

It is important that this transformation not severely impact the resource or recurrence constraints of the loop, since these are important to subsequent, performance-critical transformations such as modulo scheduling. While it is possible that the need to eliminate the inefficiencies of non-buffered execution outweighs a possible resource or recurrence degradation, this balance might vary from architecture to architecture. Height-reducing transformations such as those reflected in (d) help to ensure a benefit. Here, in particular, we see *expression reassociation* (allowing the upward motion of the predicate define) and *partial dead-code removal* (the store of 0 to r2 can now execute in parallel with the increment of r2, since they execute on mutually exclusive predicates). In addition, the loop-back branch is transformed to a special counted loop form, eliminating the inductor, and directing instruction fetch to fall out of the loop buffer on the last iteration. In this case, such optimizations are able to maintain the recurrence constraint of the inner loop even when the outer loop is collapsed into it. Provided that the inner loop schedule can accommodate the two extra instructions, the outer loop can be pulled into the inner loop with no adverse performance impact. Thus collapsing, like peeling, is able to improve performance and efficiency by keeping execution within the loop buffer for longer periods of execution time. Predicated loop collapsing was applied to 52 doubly-nested loops in the benchmark set, making them candidates for buffering.

Hyperblock side exits also present a problem for loop buffer execution, since the machine's branch resources are, appropriately, very limited. Application execution profiles reveal that, in many cases, hyperblock side exit branches are numerous but very infrequently taken. In these instances, a technique known as *branch combining* transforms several branches into a single predicated jump, guarded by a "summary predicate." The summary predicate, computed using parallel *or* compare types, is set to 1 when any exit from the loop is required; when any one of these branches would have taken, a summary jump directs execution to a "decode block" where the originally-desired control flow direction is discerned.

The IMPACT compiler's VLIW compilation support, predication, control speculation, alias analysis, profile-guided inlining, profile-directed compilation, modulo scheduling, and other traditional and ILP transformations were all applied to produce high-quality code as a basis for these experiments. Execution profiling is critical, as it helps to distinguish important paths from those less frequently executed, a necessity for many ILP techniques, including hyperblock formation. Profiling also directs function inlining, which is performed to enhance formation of loop regions, since loop regions in our implementation may not contain calls to subroutines. In the experiments that follow, profile-guided inlining was performed up to an estimated limit of 50% static code expansion. Pointer analysis is important for disambiguating pointer-based loads and stores in C code, and is important to both optimization and instruction scheduling. Finally, it is necessary for the compiler to be able to understand the relations among predicates to perform effective optimization on and around predication. These techniques form the foundation of a useful predicated ILP compiler, and were all applied in the experiments that follow.

4. Predication model

Most modern implementations of full predication are descended from the HP Labs HPL-PD design [14], a generalization of the model developed for the Cydrome Cydra 5 [15]. The IMPACT model [16], one derivative, specifies the predicate define:

$$(\mathtt{p}_g)\ \mathtt{p}_{d0}\_t_0,\ \mathtt{p}_{d1}\_t_1 = (src_0\ \mathbf{cond}\ src_1)$$

Here the guard predicate ($\mathtt{p}_g$) and comparison (src_0 **cond** src_1, where **cond** can be $=$, $\neq$, $>$, etc.) are shared by the operation's two predicate computations, which potentially write to predicate registers $\mathtt{p}_{d0}$ and $\mathtt{p}_{d1}$. Table 2 shows the functions implemented by the various predicate define types, indicating how each destination $\mathtt{p}_d$

Table 2: Predicate definition truth table.

p_g	C	ut	uf	ot	of	at	af	ct	cf
0	0	0	0	-	-	-	-	-	-
0	1	0	0	-	-	-	-	-	-
1	0	0	1	-	1	0	-	0	1
1	1	1	0	1	-	-	0	1	0

is updated based on the computation type t, guard predicate p_g and condition C. In the table, a '-' indicates that no update occurs. The two types required for if-conversion are the unconditional (ut/uf) type, which computes simple conditions, and the or (ot/of) type, which is used to compute compound conditions (*i.e.* `(x<0)||(x>3)`). Each operation possesses a guard predicate operand (p_g). Lacking the or-type predicate and the ability to predicate all operations, most partial implementations of predication are limited to eliminating simple control flow graph hammocks and diamonds, but cannot handle general control flow.

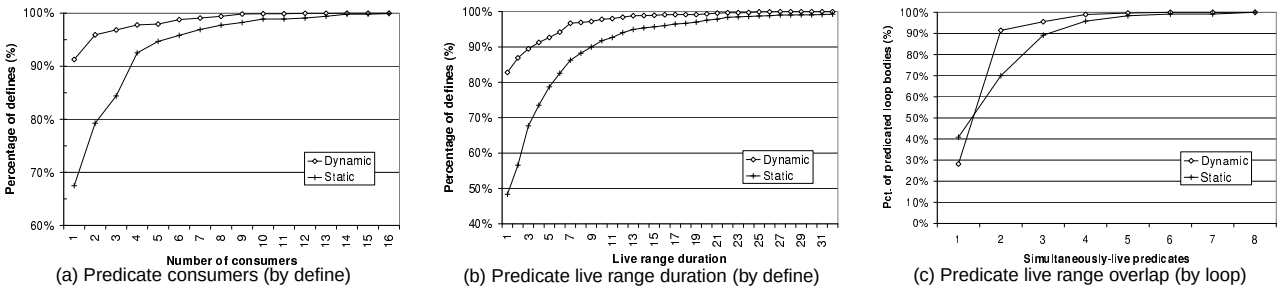
Full predication is typically based on a register-storage model, making it easy to represent and manipulate in the compiler. Implemented in hardware, it involves the addition of a new register file, the addition of new bypass logic for predicate values, the modification of existing bypass logic to allow squashing of nullified operations, and, perhaps most costly, the addition to each operation of a guard predicate operand. The IMPACT model, like the Itanium model [6], consumes 41 bits per operation—today unacceptable in an embedded domain. Bits in embedded operation encodings are at a premium, as memory size is limited and ILP techniques critical to performance in media applications need many registers to express enough parallelism for wide-issue cores. Adding a field for a guard predicate reduces addressable general register space—providing only eight predicate registers takes three bits per operation, cutting the addressable general register space in half (assuming a three-operand opcode format). Thus, while we use the flexible IMPACT predication scheme during most of the compilation process, a similarly general (*i.e.* allowing if-conversion) but less costly scheme must be implemented in hardware.

In the context of a VLIW, the compiler has the benefit of knowing to which execution units operations will issue, and in what order. Several clustered architectures already take advantage of this by requiring the compiler to partition operations into connected subgraphs for execution in separate clusters of a sparsely-connected machine [1, 17]; others have applied this principle to avoid writing dead values back into the register file after forwarding has occurred in the pipeline, as a power optimization [18]. Taking these ideas a step further, it is possible to conceive of a general predication system wherein predicates are explicitly "source-routed" from predicate defining operations directly to the units and, indeed, the very operations they are to control. This concentrates changes to the instruction set in the predicate defines themselves, rather than in the consumers (as, for example, a register-based system does not). Additionally, the hardware required to implement such a system would be easier to incorporate into an existing pipeline than would be a predicate bypassing network. Such an approach, however, is only practical when the number of consumers per predicate is small, or when several consumers can be grouped together in an easily addressed unit; otherwise, a stifling number of predicate defines are required.

4.1. Benchmark predication characteristics

Examining the predication support required in the selected benchmarks guides selection of an appropriate representation. Loop kernels, and in particular, straight-line modulo pipelined kernels, are of primary concern since these are the code regions targeted for inclusion in the loop buffer and therefore the regions in which effective predication support is critical. Having compiled the benchmarks for an implementation of the IMPACT model with eight predicate registers, using aggressive traditional and ILP optimizations, control speculation, modulo scheduling, and the techniques described in the previous section, we examine how extensively the benchmarks used the freedom of full predication and how that model could be reduced (within this application domain) to an equally effective but less costly form.

The studied applications contain 564 modulo pipelined candidate loops, of which 122 use predication. Figure 3 shows three metrics of the predication applied in the benchmarks; "static" metrics refer to physical instances of operations, while "dynamic" metrics refer to the number of times the machine encounters an operation while running the benchmark input. Figure 3(a) shows the cumulative distribution of the number of consumers per predicate define (*i.e.* 97% of predicates generated guard three or fewer operations). More than 8% of predicate defines issued have multiple consumers; about 2% of predicate defines issued have more than four, and some have as many as 16. Figure 3(b) illustrates another problem: over 3% of predicate live ranges last more than 8 cycles, indicating that regenerating nullifications at a later time for multiple consumers may increase general register pressure as well as operation count (since comparison source operands would need to be preserved). Examining the direct-control system in this light, it appears that, while in the common case performance could be reasonable, degradation in the multiple-consumer case is catastrophic. Clearly, a one-to-one or one-to-two operation nullification scheme is impractical in most cases, and encoding more than that in a 32-bit operation format would be a significant challenge. Likewise, nullifying all operations in a slot for a given number of cycles, a simple addressing scheme, does not make best use of the machine.



(a) Predicate consumers (by define) (b) Predicate live range duration (by define) (c) Predicate live range overlap (by loop)

Figure 3: Media application predication (cumulative distributions)

A compromise approach places a single bit in each operation that indicates whether or not it is "sensitive" to its guard predicate. Predicate defines then set a single predicate for each slot which has the power to nullify all subsequent sensitive operations in that slot. This scheme is efficient as long as the number of simultaneously live predicates is less than the number of slots available in the machine, and as long as the scheduler has enough freedom to order dependent uses of the same predicate into one or a few slots. To test this hypothesis, the code was prepass- and modulo-scheduled given infinite virtual predicate registers, and then colored to eight physical predicates (no spilling of predicates was required). Thus, given a general (uniform) issue machine, eight predicated slots are sufficient to implement all the program's predication. As figure 3(c) shows, only four predicates are sufficient to implement 99% of the dynamic iterations of the 122 predicated loops. Where insufficient slots exist to maintain the live predicates, either extra predicate defines must be inserted to regenerate predicate values in split live ranges, or scheduling and optimization aggressiveness must be reduced. Since such intervention appears largely unnecessary in these benchmarks, this model effectively balances implementation cost and generality.

4.2. Slot-based predication

Figure 4 shows one possible 32-bit encoding of the predicate defining operations. Commonly used pairs of destination types are selected as units, much as in Itanium. With another two bits (perhaps using more of the opcode space) it is possible to encode all the destination type pairs, but the extra combinations are only infrequently used. The predicate defines can be relatively complex because the destination encoding size is small, given that there are only eight slots in the machine. It is likely that a VLIW beyond this size would have clusters, each of eight slots or less, so this scheme generalizes easily in that direction. In that case, a predicate define would control only slots in its own cluster. Since there is no constant-value predicate (usually $p0$) in which to "dispose of" unwanted results, a single-destination predicate define specifies the same slot twice. In such a case, the second definition is defined to have no effect. The predicate define, aside from writing to slots as opposed to standard predicate registers, is fairly ordinary. Under HPL-PD / IMPACT predicate define semantics (Table 2), update values are independent of the previous value of the destination predicate; thus, define evaluation does not require the previous destination value.

The more novel part of this scheme is what happens with the predicate values themselves. Rather than being placed in a predicate bypass network and being written into a predicate register file, as in a traditional implementation, predicate updates here are sent directly to the slots they control. Since the machine is explicitly scheduled and the computation of the predicate value is of known duration (presumably one cycle), predicate updates are simply sent from the generating unit to the consuming slot, where they modify the predicate visible to subsequent issuing operations. An update, sent only when the appropriate entry of Table 2 is zero or one, specifies the new value to be written and implies that a write should take place. It is allowable for two defines to write simultaneously to the same slot as long as they write the same value, as can occur with or-type defines. The compiler prevents two defines which could write 0 and 1 to the same slot from being scheduled together. Under this scheme, each slot has one "standing predicate" which remains the same until reset by a predicate define; operations in that slot with their "predicate sensitivity bit" set are guarded on their slot's standing predicate. Since the statically-defined ordering of bundles enforced by VLIW issue maintains the dependences between predicate defines and predicate consumers, register names and scoreboarding of predicates are unnecessary.

Figure 4 shows a conceptual diagram of the pipeline's predication "harness." The figure depicts the predication-related features of three of the machine's eight slots: here, slots 0 and 1 both generate and consume predicates, and slot 7 consumes them. At the top of the diagram are decoded instruction fields which control the predication features. The p bit indicates whether or not an operation is sensitive to nullification by its guard predicate. An operation is nullified (or the guard predicate is considered to be 0, in the

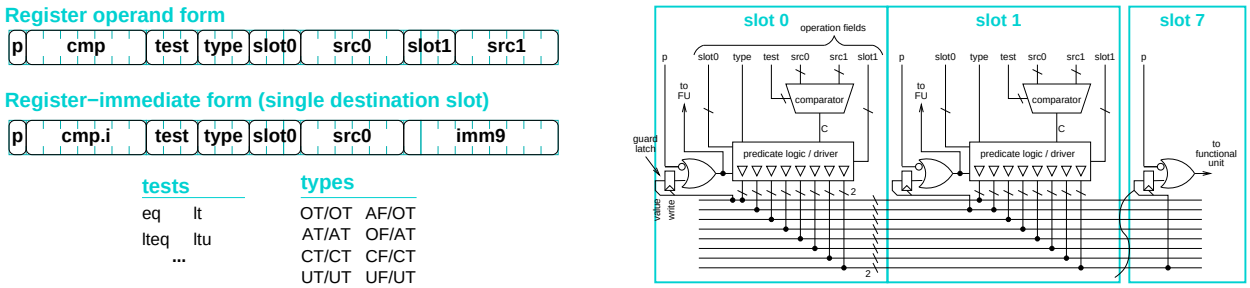


Figure 4: Predicate defines and predication support features

case of the predicate defining operations) only when the p bit is 1 and the computed predicate, stored in the guard latch is 0. Predicates are distributed on a 16-bit bus, which contains a value line and a write line for each slot. The bit on the value line is latched when the write line is active. Both these lines are driven by tri-state drivers in each predicate-generating unit; during an update, the predicate define unit activates the target slot's write line and drives the target slot's value line to the desired value.

4.3. Code generation

In this approach, a slot's predicate remains set to a given value until it is reassigned, but nullifies only sensitive operations; the compiler is thus free to intersperse predicated and non-predicated operations in a given slot, but only one predicate is available in each slot at any given time. Predicate defines and predicate consumers must be scheduled such that live ranges, now tied to the issue slot of the consumer, do not interfere. This differs little from allocating the predicates to eight registers; however, two new constraints appear. First, each physical predicate, associated with one slot, can guard only one operation per cycle. Second, and more seriously, in a non-uniform machine, in which different slots perform different operations, it may be necessary to replicate a logical predicate to multiple slots if it has different types of consumers. To assist in providing the necessary control, most predicate defines can supply two slot predicates, as indicated in Figure 4. As indicated in the empirical studies above, these new constraints do not appear to be serious limitations for the studied applications and architecture. The complexity of assigning predicated instructions to slots and of providing the necessary predicate defining instructions increases significantly with the asymmetry of the machine and, in the compiler, requires tighter coupling of operation scheduling to mechanisms traditionally part of the register allocator.

As is usual with registers, it is beneficial to keep predicate lifetimes as short as possible to enable best reuse of resources. One technique that helps to do this in the case of these predicates is *predicate promotion*, the removal of a guard from an operation that may safely be executed when the predicate is false (although the result is unneeded) [13]. By removing the predicates from all but those that absolutely require guards, the compiler reduces the stress on this critical resource. Given these efforts, 21.5% of dynamic operations in predicated loops are sensitive to predicates (9.9% in all bufferable loops). Compiler and architectural support for general speculation and a generous supply of functional units are critical components of this promotion-based model.

Table 3: Buffer management operations

Operation	Functionality
`rec_cloop` *buf_addr, num, count*	Buffer *num* subsequent operations at address *buf_addr*, if not already in buffer, and commence *count* iterations. Fall through to operation after `br_cloop`.
`rec_wloop` *buf_addr, num*	Buffer *num* subsequent operations at address *buf_addr*, if not already in buffer, and iterate until `br_wloop` fails. Fall through to operation after `br_wloop`.
`exec_cloop` *buf_addr, count*	Execute the loop buffered at *buf_addr count* times. On exit, continue after `exec_cloop` operation.
`exec_wloop` *buf_addr, count*	Execute the loop buffered at *buf_addr* until its `br_wloop` falls through. On exit, continue after `exec_wloop` operation.

5. Loop buffers

A variety of loop buffer implementations exist in the DSP market. We choose to implement the loop buffer as a compiler-managed cache, mapped architecturally into the instruction address space, but residing on-chip in a physically different location. The compiler manages the buffer as a resource, scheduling loop bodies into segments of the buffer as required. As alluded to in Section 6, the goal of scheduling loops into the buffer is to minimize the total number of bundles fetched from the global memory.

The compiler controls buffering by means of the four operations shown in Table 3. To buffer a counted loop, for example, the compiler prefaces the loop with the (branch unit) operation `rec_cloop` *buf_addr, num, count*. This instructs the instruction fetch unit to start buffering the subsequent *num*-operation loop at the buffer offset *buf_addr*, and to execute that loop *count* times. During the first loop iteration, the loop is both being executed and being stored into the buffer; subsequent iterations are executed directly from the loop buffer. When the loop is finished executing in the buffer, control must be returned to the global fetch mechanism. Since the size of the loop body and the address of the initial `rec_cloop` operation are known, this

address is easily computed—the following bundles could even be prefetched by the instruction fetch logic if desired. The `rec_wloop` operation functions similarly for loops of unknown duration, but does not prepare the loop buffer to correctly predict loop exit like the `cloop` version.

The `exec_[cw]loop` operations execute a loop known already to be stored in the buffer, returning to the operation after the `exec_[cw]loop` on exit. These enable a buffered loop to be reused from different locations in the code, almost like a procedure call, as a code size optimization.

With a small amount of additional hardware, a table can be created which maps buffer offsets of active loops to the address of their `rec` operations, achieving additional savings. Consider, for example, an outer loop containing a collection of buffered loops, all small enough to cohabitate in the loop buffer. When one of the inner-loop `rec` operations is encountered for the second time, the table will indicate that the loop buffered from that address is known to be intact in the buffer, so re-recording of the loop's operations will not occur, but the loop exit will still fall through to a location *num* operations later, after the end of the loop's image in global memory. It is important to note that the loop buffer here is not operating as a hardware cache, as the compiler is responsible for explicitly controlling its population; the hardware is simply given a small memory to avoid useless work. An example of the operation of the loop buffering system is given in the next section.

6. Code example

To demonstrate loop buffering, we examine the function Post_Filter() from the Global System for Mobile Communications (GSM) Enhanced Full-Rate (EFR) speech decoder *g724dec* [10]. After inlining and transformations, this function accounts for 49% of *g724dec*'s execution cycles on the target machine—overall buffer performance thus depends heavily on this function (see Section 7 for machine description). Figure 5(a) shows a control-flow graph of the function's 13 loops after the transformations of Section 3. Backedges are labeled with their traversal weights, per iteration of the outer loop, which has four iterations. After if-conversion of "C" and "J," both of which contain internal control flow, the twelve inner loops are modulo scheduled.

To the right of the function's control-flow graph are demonstrations of buffer scheduling for three different buffer sizes: 16, 32, and 64 operations. Figure 5(b) shows an "execution trace" of the four outer loop iterations, indicating the times at which recording and replay take place (time runs vertically through each iteration). To the right of this trace is a "buffer trace" indicating what loop is active and which loops reside in the buffer at a given time. Any horizontal slice yields the contents of the buffer at that time. Both traces are aligned with the control flow graph at the left; columns to the right of the buffer show the number of each loop's iterations the compiler was able to schedule for buffer issue. The compiler must choose locations for each buffered loop, such that needed loops will not conflict with each other.

The 16-operation buffer's success is limited because its size is inadequate for all but three loops, and these displace each other over the execution of the outer loop. Each time the `rec_cloop` prefacing a bufferable loop is encountered, recording of the loop is performed while the first iteration executes through global fetch. As calculated from the "buffered iterations" column, the 1056 operations issued from the loop buffer represent only 1.23% of Post_Filter()'s 85869 loop and non-loop operations. For the 32-operation case (c), several more loops fit within the buffer, but because the sum of the smallest operation-count loop ("E") and many other loops is greater than 32, by the time "E" is reached again in the control flow graph, its instructions are no longer intact in the buffer (*i.e.* the compiler has dictated its replacement by "H").

The benefits of targeted hyperblock formation are clearly seen in (d), where approximately 99% of loop execution (or 98.2% of Post_Filter()'s total instruction issue) occurs from the 64-operation buffer. Without the techniques described in Section 3 , the performance of this function would be significantly degraded because the two highest iteration count regions would no longer be loops suitable for buffering, and would thus not only incur branch penalties, but also cause greater power consumption. The power benefit of this mechanism will be further explored in Section 7.2. In (d), both loops "E" and "F" were detected by the compiler to be compatible in size with the 49-operation loops "C" and "J." "F" was selected for residence over the entire period of function execution because its recording overhead (14 operations) is greater than that of "E" (12 operations). "F" is still prefaced with a `rec_cloop` operation so it is recorded into the buffer when the outer loop first executes. In subsequent outer loop iterations, the hardware table will indicate that "F" is still in the buffer, and issue of the `rec_cloop` operation will cause loop execution to begin from the buffer without re-recording.

7. Experiments

Our experimental machine is an 8-wide unified VLIW with resources loosely modeled after the TI 'C6x series microprocessors. It is assumed that operation bundles are stored in memory in a compressed format [19] in which NOPs consume no space (as in TI 'C6x), and each operation is assumed to be 32 bits in length. The processor has eight integer ALUs, two of which can issue integer multiplies; three memory units; one branch unit; two floating-point units; and four units capable of generating predicate values. Figure 6 shows the fixed assignment of functional units to slots. Arithmetic operations have a latency of 1 cy-

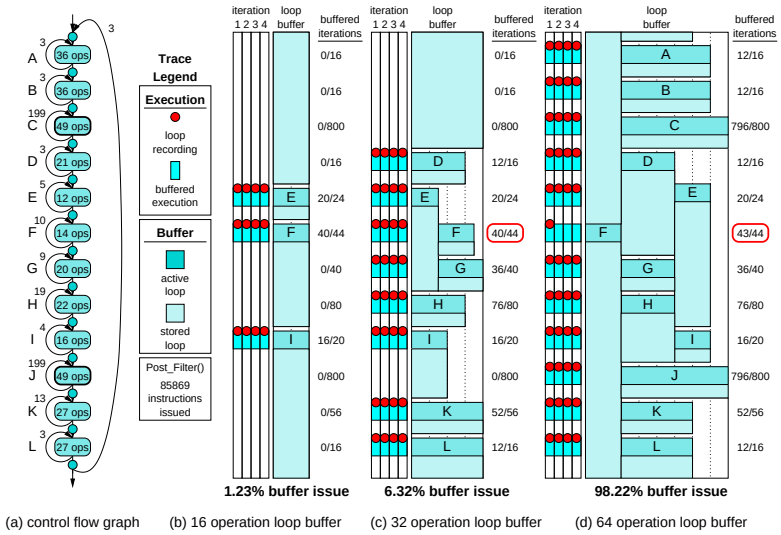


Figure 5: g724dec Post_Filter() loop control flow graph and buffer content traces

0	1	2	3	4	5	6	7
Imul/F Pred	Imul/F Pred	Ialu Pred	Ialu Pred	Ialu Mem	Ialu Mem	Ialu Mem	Ialu Br

Figure 6: VLIW issue slots for modeled architecture

cle; multiplies, 2 cycles; divides, 8 cycles; loads, 3 cycles; and floating point arithmetic, 2 cycles. Sixty-four (64) integer registers are provided. General control speculation is supported by providing all potentially excepting instructions except for stores with a speculative form. The architecture implements the predication scheme of Section 4 with its four predicate-generating units; all slots are capable of receiving predicates.

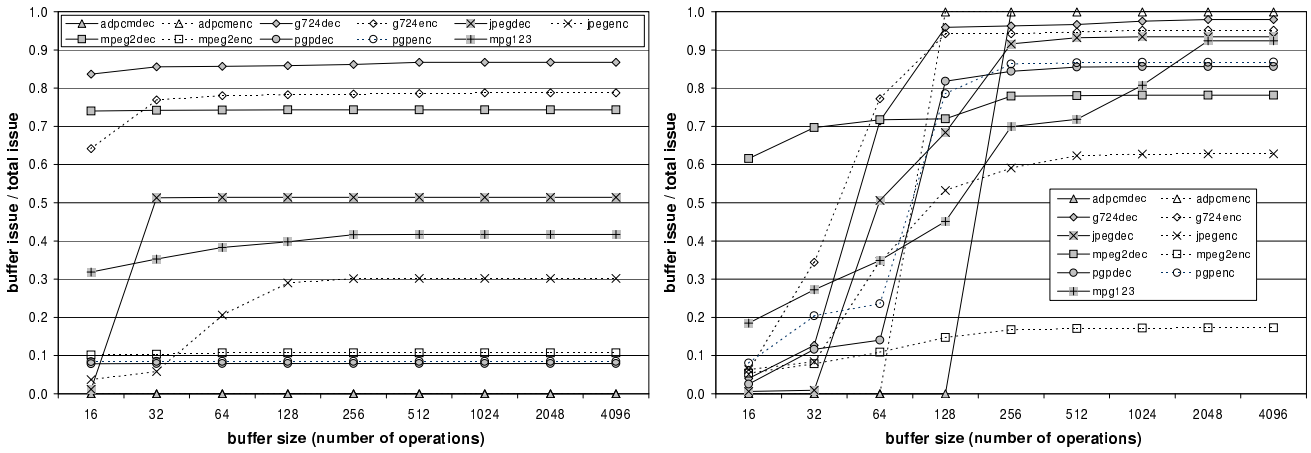
7.1. Effectiveness of transformations

The direct measurement of branch overhead reduction and power savings depends on microarchitectural details below the level we choose to model. Loop buffers today are a ubiquitous feature in DSP, existing in many implementations from hardware buffers capable of holding a few instructions in a single loop, to complex buffers holding multiply-nested loops, to what is effectively a set of pointers into a software-managed instruction cache which implements buffering for very large loops. Likewise, DSPs have a wide variety of pipeline lengths and branching implementations, often including delay slots. In general, though, one thing constitutes an improvement across all architectures, regardless of these variations: getting more of the program into a loop buffer. This, therefore is the metric we choose to measure the effectiveness of our compiler transformations in fitting loops into the simple buffer model described in Section 5.

Two compilations were performed for each benchmark: one applied only traditional compiler optimizations (*i.e.* no predication and no loop collapsing), and the second aggressively applied control transformations (hyperblock formation, unrolling, peeling, and collapsing) intended to enhance opportunities for instruction buffering. In both cases, profile-guided selective inlining was performed with up to 50% code expansion, modulo scheduling with modulo variable expansion was performed, and loop bodies were scheduled into the loop buffer by the compiler, given a control flow profile. Figure 7(a) shows, for each of the benchmarks, the fraction of instruction accesses that are satisfied by the loop buffer when only traditional optimizations are applied. Although the benchmarks contain some small loops with no internal control flow, the benefit clearly saturates at a small buffer size. Figure 7(b) shows the same metric for benchmarks after aggressive transformation, in which a much greater proportion of execution is captured. The *adpcm* benchmarks resolve for the most part to a single predicated loop which, once scheduled into the loop buffer, accounts for over 99% of instruction issue. After aggressive control transformation, even much more complex benchmarks such as the *g724* codec issue between 94 and 96% of instructions from the loop buffer.

In light of the code example given in Section 6, it is of interest that a majority of *g724dec*'s execution occurs in the loop buffer from the 64-operation size onward, which clearly confirms that the success of Post_Filter() in the buffering mechanism has a great impact on the overall benchmark's outcome. An additional increase of about 20% in loop buffer issue is realized for *g724dec* at the 128-operation buffer size because important loops in functions other than Post_Filter() are then buffered.

Of the benchmarks for which less than three-quarters of instruction issue is captured by the loop buffer, *mpeg2enc* is the most problematic. The MPEG-2 encoder contains many large, highly nested loop structures which only iterate several times. Though loop collapsing simplifies loop



(a) Traditional code optimization only

(b) With hyperblock transformations

Figure 7: Percentage instruction issue from loop buffer

structures in many applications, *mpeg2enc*'s nested loops do not meet our criteria for small outer loop size and moderate inner loop iteration count. Hyperblock transformations do, however, almost double the fraction of *mpeg2enc* instructions issued from the buffer, but because loop buffering is most effective for high-iteration count or small, high instance count loops, *mpeg2enc* is unlikely to achieve 80-90% loop buffer issue rates, regardless of the code transformations applied.

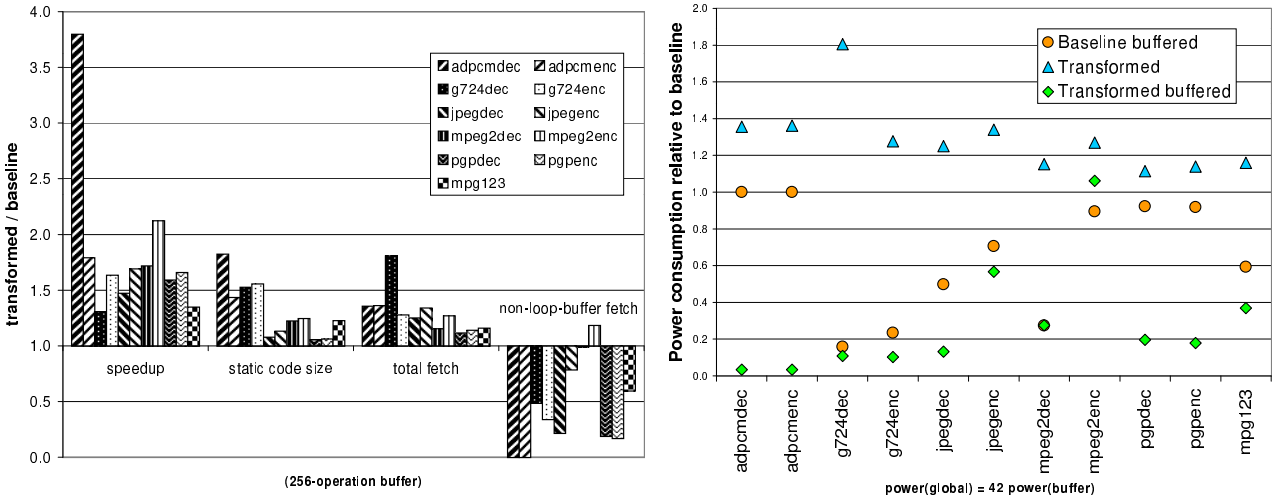
Similarly, predication and loop transformation techniques realize a twofold buffer issue increase for the *jpegenc* benchmark, but this final value saturates at 63%. The JPEG encoder was found to have significant numbers of inner-nest loops for which the iteration counts were generally small, but varied across different loop invocations. Peeling these loops into an outer hyperblock could significantly increase dependence heights, but if buffer eligibility were to be given priority over dependence height and function code size expansion, a much more significant proportion of *jpegenc*'s instructions could be merged into single-hyperblock loops which issue from the loop buffer. Likewise, collapsing could be applied, but since these outer loop bodies contained more code than their inner loops, this would be detrimental to performance. Although buffering results are moderate relative to other benchmarks, the compiler heuristics behaved correctly in leaving the loops unmodified.

The buffer performance of our MPEG-2 Layer 3 audio decoder, *mpg123*, struggles except for very large (2048-operation) buffer sizes primarily because its execution time is concentrated in functions with small trip count loops, which, for optimal performance, must all remain in the loop buffer simultaneously. Additionally, this benchmark contains a number of large loops which were very effectively modulo scheduled, but which require four modulo variable expansions, thus increasing their code size. When modulo scheduled with a rotating register algorithm, the size of these loops was decreased, indicating that buffer performance could likely be further improved through use of architected rotating registers.

7.2. Effects of Loop Buffering

While the loop buffering results are encouraging, it is important to consider other effects of the enabling transformations. Figure 8(a) draws comparisons between the ILP-transformed code and traditionally optimized programs. Clearly ILP transformations trade code size for improved performance. An average program speedup of 1.81 is achieved with our techniques, but as shown by the second grouping, static code size increases significantly relative to traditionally optimized forms. Additionally, while the number of instruction *bundles* issued does not increase, the total number of *operations* fetched ("total fetch" in Figure 8) does.

Finally, we examine the net effects of buffering and transformation on the benchmark set. Figure 8(a) shows that for all but one benchmark (*mpeg2enc*), the transformations applied for loop buffering significantly decrease the amount of code which is fetched from standard instruction memory. Cacti 2.0 [20] power simulation mechanisms were used to quantify the impact of this shift in fetch location on processor power consumption. While DSP architectures traditionally had separate instruction and data memories, wider, more general architectures use a single, large, unified on-chip storage space [3]. While memory power is obviously highly dependent upon design style, it is not uncommon to see a linear increase in power consumption with size. Cacti results for 0.13μm technology indicate that fetching an operation from a single-port, 256-operation buffer (assuming 32-bit operations) consumes 41.8 *times* less power than a fetch from a 512KB, 2 read/write port, non-cache memory. This result provides a means of assessing the power implications of the proposed techniques.



(a) Performance, code size, and fetch count

(b) Estimated instruction fetch power (normalized)

Figure 8: Hyperblock and loop-transformed code vs. traditional optimizations

Since use of a 256-operation buffer allows incorporation of large op-count, high iteration loops, many loops will consume enough cycles to reasonably assume that clock gating could be effective on elements of the global instruction fetch hardware while the loop buffer was being accessed. Figure 8(b) shows the estimated power consumption of instruction fetch, normalized to buffer-less issue of traditionally-optimized code. While a 256-entry loop buffer operating on non-transformed code reduces instruction fetch power by an average of 34.6% ("Baseline buffered"), the proposed transformations reduce it by 72.3% ("Transformed buffer") relative to the same non-buffered, non-transformed baseline.

7.3. Limitations and future work

While experimental results, some of which were too detailed to include in this format, demonstrated that binding predicates to particular slots did not significantly affect performance or code size in this benchmark set, the implementation of compiler techniques to best manage these intricacies in the more complex benchmarks such as *mpeg2enc* and *jpegenc* is continuing.

As the architectural description indicated, the low-overhead predication scheme adds a 1-cycle critical path from the output of the predicate-generator to the predicate-squash input of the controlled functional unit, as well as a system of global busses for communicating predicate updates. These are not likely to cause a design constraint unless the processor is already clustered. Should it become a problem, the predicate harness could itself be clustered, either to the same degree as the VLIW, or to an even greater degree. While this further complicates the compiler's job, it seems consistent with other existing work in clustering [1].

Finally, the source-routing of predicates by the simple scheme presented could be extended in more complex and elegant directions–possibly to perform more complex operations, such as queuing a predicate to become active at some future time (easing the liveness constraint), or possibly to include some limited-lifetime values traditionally stored in general purpose registers. The current scheme, while demonstrably sufficient for the benchmarks illustrated, may require greater than the optimal number of predicate defines to guard general predicated computation.

8. Previous work

Alternative loop buffering mechanisms were described in Section 2. These hardware features are widely implemented in the DSP community, but their implementation and optimal application have not been explored extensively in academic publications. [21] studied the 31-instruction loop buffers in the Lucent DSP16000 using a set of small DSP kernels and demonstrated a 24.8% decrease in execution cycles without compiler transformations and a 32.7% decrease in execution cycles with compiler transformations. Among these transformations were function inlining and the use of the conditional execution support of the DSP16000 to include loops with simple control diamonds.

Various techniques have been studied for incorporating predication into an architecture without increasing the per-instruction encoding size, at least for non-predicated operations. Probably the most prevalent of these is the conditional move [22], a single "predicated" instruction which operates on the Boolean value stored in a general-purpose register, to select from among speculatively-generated results. Unfortunately, operations which may not be speculated must receive special conditional move handling which requires more than a single instruction. [22] showed a 44% dynamic code size increase for this technique relative to full predication, which makes it questionable for inclusion in

an embedded processor. Other papers [23] have proposed adding guard predicates as instruction prefixes, but this is not in general applicable to the fixed-operation-encoding-size VLIW model typical of the latest DSPs. PA-RISC provides *instruction nullification* to allow an instruction to conditionally nullify one subsequent operation, but this technique lacks the generality necessary for if-conversion, and consumes encoding space in many operations.

9. Conclusions

Elimination of control flow overhead is a key component of high-performance compilation on processors intended for the media and telecommunications segment. We demonstrated techniques supporting the elimination of general internal control flow using full predication, and techniques which improve the utilization of a simple, one-level loop buffer by transforming control flow into a more regular form. The implicit structure of the VLIW was then used to develop a mechanism which provides the benefits of full predication without a predicate register file or a predicate bypass network. Results achieved for an 8-wide VLIW in a high-performance compilation environment indicate a 137.5% increase in the percentage of instructions issued out of the loop buffer, relative to traditional compilation techniques, together with an average speedup of 1.81. Finally, the transformations applied were shown to more than double the instruction fetch power benefit of loop buffering, allowing loop buffering to reduce instruction fetch power by 72.3% in the modeled machine. These results encourage further exploration into models of communication within the VLIW pipeline, which may have the potential to reduce register pressure, save power, and simplify hardware.

Acknowledgments

This work was supported by the Semiconductor Research Corporation under the grant "Memory Efficient EPIC/VLIW Architecture (ID 785)" and by a grant from StarCore. John Sias and Hillery Hunter were supported by fellowships from the IBM Centre for Advanced Study and the National Science Foundation, respectively. We thank the IMPACT contributors for their work on the compiler, particularly Matthew Merten, who recently enhanced modulo scheduling, and the anonymous reviewers for their comments.

References

[1] J. Sánchez and A. González, "Modulo scheduling for a fully-distributed clustered VLIW architecture," in *Proc. 33rd Int'l Symposium on Microarchitecture*, pp. 124–133, Dec. 2000.

[2] B. R. Rau, "Iterative modulo scheduling: An algorithm for software pipelining loops," in *Proc. 27th Int'l Symposium on Microarchitecture*, pp. 63–74, Dec. 1994.

[3] StarCore DSP Technology, *SC140 DSP Core Reference Manual*, June 2000.

[4] K. Ebcioglu, "A compilation technique for software pipelining of loops with conditional jumps," in *Proc. 20th Int'l Symposium on Microarchitecture*, pp. 69–79, 1987.

[5] M. Lam, "Software pipelining: an effective scheduling technique for VLIW machines," in *Proc. Conference on Programming Language Design and Implementation*, pp. 318–328, 1988.

[6] Intel Corporation, *IA-64 Architecture Software Developer's Manual, revision 1.1*, July 2000.

[7] J. C. Park and M. S. Schlansker, "On predicated execution," Tech. Rep. HPL-91-58, Hewlett Packard Laboratories, May 1991.

[8] Texas Instruments Incorporated, *TMS320C6000 CPU and Instruction Set Reference Guide*, Mar. 1999.

[9] STMicroelectronics, *ST120 DSP-MCU Programming Manual*, Dec. 2000.

[10] ETSI TC-SMG, "Digital cellular communications system; enhanced full rate (EFR) speech transcoding (GSM 06.60)," Tech. Rep. ETS 300 726, European Telecomm. Standards Institute, Mar. 1997.

[11] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, "MediaBench: A tool for evaluating and synthesizing multimedia and communications systems," in *Proc. 30th Int'l Symposium on Microarchitecture*, pp. 330–335, Dec. 1997.

[12] W. W. Hwu, R. E. Hank, D. M. Gallagher, *et al.*, "Compiler technology for future microprocessors," *Proceedings of the IEEE*, vol. 83, pp. 1625–1995, Dec. 1995.

[13] S. A. Mahlke, D. C. Lin, W. Y. Chen, *et al.*, "Effective compiler support for predicated execution using the hyperblock," in *Proc. 25th Int'l Symposium on Microarchitecture*, pp. 45–54, Dec. 1992.

[14] V. Kathail, M. S. Schlansker, and B. R. Rau, "HPL-PD architecture specification: Version 1.1," Tech. Rep. HPL-93-80 (R.1), Hewlett-Packard Laboratories, Feb. 2001.

[15] J. C. Dehnert and R. A. Towle, "Compiling for the Cydra 5," *The Journal of Supercomputing*, vol. 7, pp. 181–227, Jan. 1993.

[16] D. I. August, D. A. Connors, S. A. Mahlke, *et al.*, "Integrated predicated and speculative execution in the IMPACT EPIC architecture," in *Proc. 25th Int'l Symposium on Computer Architecture*, pp. 227–237, June 1998.

[17] P. Faraboschi, G. Desoli, and J. Fisher, "Clustered instruction-level parallel processors," Tech. Rep. HPL-98-204, Hewlett-Packard Laboratories, Dec. 1998.

[18] M. Sami, D. Sciuto, C. Silvano, *et al.*, "Exploiting data forwarding to reduce the power budget of VLIW embedded processors," in *Proc. Design, Automation and Test in Europe*, pp. 252–257, 2001.

[19] T. Conte, S. Banerjia, S. Larin, *et al.*, "Instruction fetch mechanisms for VLIW architectures with compressed encodings," in *Proc. 29th Int'l Symposium on Microarchitecture*, pp. 201–211, Dec. 1996.

[20] G. Reinman and N. Jouppi, "An integrated cache timing and power model." Summer Internship Report, COMPAQ Western Research Lab, Palo Alto, 1999.

[21] G.-H. Uh, Y. Wang, D. Whalley, *et al.*, "Efficient exploitation of a zero overhead loop buffer," in *Proc. Workshop on Languages, Compilers, and Tools for Embedded Systems*, pp. 10–19, May 1999.

[22] S. A. Mahlke, R. E. Hank, J. McCormick, *et al.*, "A comparison of full and partial predicated execution support for ILP processors," in *Proc. 22nd Int'l Symposium on Computer Architecture*, pp. 138–150, June 1995.

[23] D. Connors, J. Puiatti, D. August, *et al.*, "An architecture framework for introducing predication into embedded microprocessors," in *Proc. 5th Int'l Euro-Par Conference*, Aug. 1999.