Journal of Instruction-Level Parallelism 9 (2007) 1–26 Submitted 6/07; published 9/07

Deep Jam: Conversion of Coarse-Grain Parallelism to Fine-Grain and Vector Parallelism

Patrick Carribault[*] PATRICK@ICES.UTEXAS.EDU
Department of Computer Science, University of Texas, Austin.

Stéphane Zuckerman STEPHANE.ZUCKERMAN@PRISM.UVSQ.FR
LRC Itaca, University of Versailles and CEA DAM.

Albert Cohen ALBERT.COHEN@INRIA.FR
ALCHEMY Group, INRIA Saclay and LRI, Paris Sud 11 University.

William Jalby WILLIAM.JALBY@PRISM.UVSQ.FR
LRC Itaca, University of Versailles and CEA DAM.

Abstract

A number of computational applications lack instruction-level parallelism. This loss is particularly acute on sequences of dependent instructions on wide-issue or deeply pipelined architectures. We consider four real applications from computational biology, cryptanalysis, and data compression. These applications are characterized by long sequences of dependent instructions, irregular control-flow and intricate scalar and memory dependence patterns. While these benchmarks exhibit good memory locality and branch-predictability, state-of-the-art compiler optimizations fail to exploit much instruction-level parallelism.

This paper shows that major performance gains are possible on such applications, through a loop transformation called *deep jam*. This transformation reshapes the control-flow of a program to facilitate the extraction of independent computations through classical back-end techniques. Deep jam combines accurate dependence analysis and control speculation, with a generalized form of recursive, multi-variant unroll-and-jam; it brings together independent instructions across irregular control structures, removing memory-based dependences through scalar and array renaming. This optimization contributes to the extraction of fine-grain parallelism in irregular applications. We propose a feedback-directed deep jam algorithm, selecting a jamming strategy, function of the architecture and application characteristics.

1. Introduction and Related Work

Optimizing compilers perform a wealth of program transformations to maximize the computation throughput of modern processor architectures. These optimizations improve the behavior of architecture components, such as the memory bus (reduction of the memory bandwidth), the cache hierarchy (locality optimization), the processor front-end (removal of stalls and flushes in the instruction flow), the processor back-end (instruction scheduling), and the mapping of instructions to functional units and register banks. Yet superscalar out-of-order execution, software pipelining and automatic vectorization fail to exploit enough fine-grain parallelism when short producer-consumer dependences hamper aggressive instruction scheduling [1, 2]. Many hardware and software solutions have been proposed.

∗. While at LRC Itaca, University of Versailles and CEA DAM.

Hardware Approaches

- Simultaneous multithreading [3] is specifically aimed at the filling of idle functional units with independent computations. Yet the program must be explicitly threaded, or one may resort to automatic parallelization.
- Large and structured instruction windows also enable coarser grain parallelism to be exploited in aggressive superscalar designs [4, 5].
- Load/store speculation and value prediction can also improve out-of-order superscalar execution of dependent instruction sequences [1, 2].
- Instruction sequence collapsing bridges value prediction and instruction selection. Typical examples are fused multiply-add (FMA) or domain-specific instructions like the sum of absolute differences (SAD in Intel MMX), or custom operators [6, 7].

Software Approaches

Closer to our work, many approaches do not require any hardware support but rely on aggressive program transformations to *convert coarse-grain parallelism from outer control structures into fine-grain parallelism*. These *enabling* transformations enhance the effectiveness of a back-end scheduler (for ILP) or vectorizer. Classical loop transformations [8] may improve the effectiveness of back-end scheduling phases: loop fusion and unroll-and-jam combined with scalar promotion [9, 10] is popular in modern compilers. Several authors extended software-pipelining to nested loops, e.g., through hierarchical scheduling steps [11, 12] or modulo-scheduling of outer loops [13]. But these techniques apply mostly to regular, static-control loop nests.

Extension to loops with conditionals may incur severe overheads, and none of these approaches handle nested `while` loops. Two approaches deal with ILP beyond branches: speculative scheduling techniques coalesce consecutive basic blocks, while *software thread integration* merges instructions coming from independent procedures to increase ILP.

Speculative Scheduling. Trace scheduling [14] can increase the amount of fine-grain parallelism in intricate acyclic control-flow, but its ability to convert coarser-grain parallelism is limited. It aims at using the trace of the program to group consecutive basic blocks and consider these blocks as one. A function is divided intro traces representing the frequently-executed paths. Side entrances and side exits are allowed in these traces. Instructions are scheduled within a trace ignoring branches. But this has a major drawback: the implementation complexity incurred by the need to maintain correct program execution after moving instructions across basic blocks (bookkeeping).

Superblock scheduling [15, 16] is also a technique for exploiting ILP across basic-block boundaries. A superblock is a trace which has no side entrances: side entrances are removed thanks to tail duplication [17, 18]. Formation of superblocks can be directed thanks to dynamic feedback from profiling and static analysis [16].

Software Thread Integration. Independently, *software thread integration* (STI) [19, 20] is designed to map multithreaded applications on small embedded devices without preemptive multitasking operating systems. STI proceeds with the static interleaving of independent threads into a single sequential program, considering arbitrary control flow, including procedure calls. This technique has recently been proposed to exploit coarse-grain parallelism on wide-issue architectures [21]: it

statically interleaves several procedures calls (from different procedures or not). Finally, STI has been extended to iterative compilation [22].

Yet STI does not allow any dependences between the threads being statically interleaved, it only provides rough support for nested conditionals (decision trees) and `while` loops, and requires the manual choice of the procedures to integrate.

Contributions. This paper presents a new program optimization, called *deep jam*, to convert coarse-grain parallelism into finer-grain instruction or vector parallelism.[1] Deep jam is a recursive, generalized form of unroll-and-jam; it brings together independent instructions across irregular control structures, breaking memory-based dependences through scalar and array renaming. This transformation can enhance the ability of a back-end optimizer to extract fine-grain parallelism and improve locality in irregular applications. Deep jam revisits STI to (statically) interleave multiple fragments of a *single* sequential program, associating it with optimizations for decision trees, scalar and array dependence removal, and speculative loop transformations. We show that deep jam brings strong speedups on four real control-intensive codes, allowing idle functional units to be fed with independent operations, with a low control overhead.

This paper is organized as follows: Section 2 introduces the primitive jamming transformations of the control-flow, including scalar and array renaming, and states a first deep jam algorithm. Section 3 explains the criteria of jamming and variations to adapt to dynamic execution profiles. Section 4 integrates all these analyses and transformations in a generic deep jam algorithm and proposes hints to design a practical algorithm, reducing the parameter space induced by the generic one. Section 5 describes four real applications and their performance inefficiencies, then shows how deep jam can achieve good speedups.

2. Jamming Irregular Control With Data Dependences

Figure 1 shows a single-stage basic control-flow transformation performed by deep jam. In this example, the outer loop cannot be fused with other loop to increase the ILP with control- and data-independent instructions. Moreover, the scalar variable `a` produces many intra-loop — unlabeled or 0-labeled edges — and loop-carried dependences — edges with positive distance labels [8]. To improve performance, let us first unroll the loop (step b). The main purpose is to merge (or jam) structures with similar control-flow and independent instructions. For that, the unrolling of the outer loop exhibits two pairs of identical structures: `if`/`if` and `while`/`while`. But the data dependences on `a` hamper the code motion needed to group these structures. Therefore, a partial renaming is mandatory: this is a SSA-like renaming, but only one assignment per structure is needed (step b). Then, step c matches pairs of identical control structures by moving the second `if` above the second `while` loop, then step d jams `if` conditionals and `while` loops pairwise (respectively).

This can be seen as a generalized unroll-and-jam [8] for irregular control, including non-loop structures. Performance improvements come from the execution of larger basic blocks with increased IPC: when conditions `p1` and `p2` (resp. `q1` and `q2`) hold simultaneously, instructions coming from two subsequent iterations of the outer `for` loop may be concurrently executed.

1. Deep jam was first presented in a conference [23]. This longer version describes a compilation algorithm at much greater depth, and draws a pragmatic roadmap to implement deep jam while optimizing the profitability of its steering heuristics. It also reports on the successful optimization of two additional applications, including the SPEC benchmark Gzip.

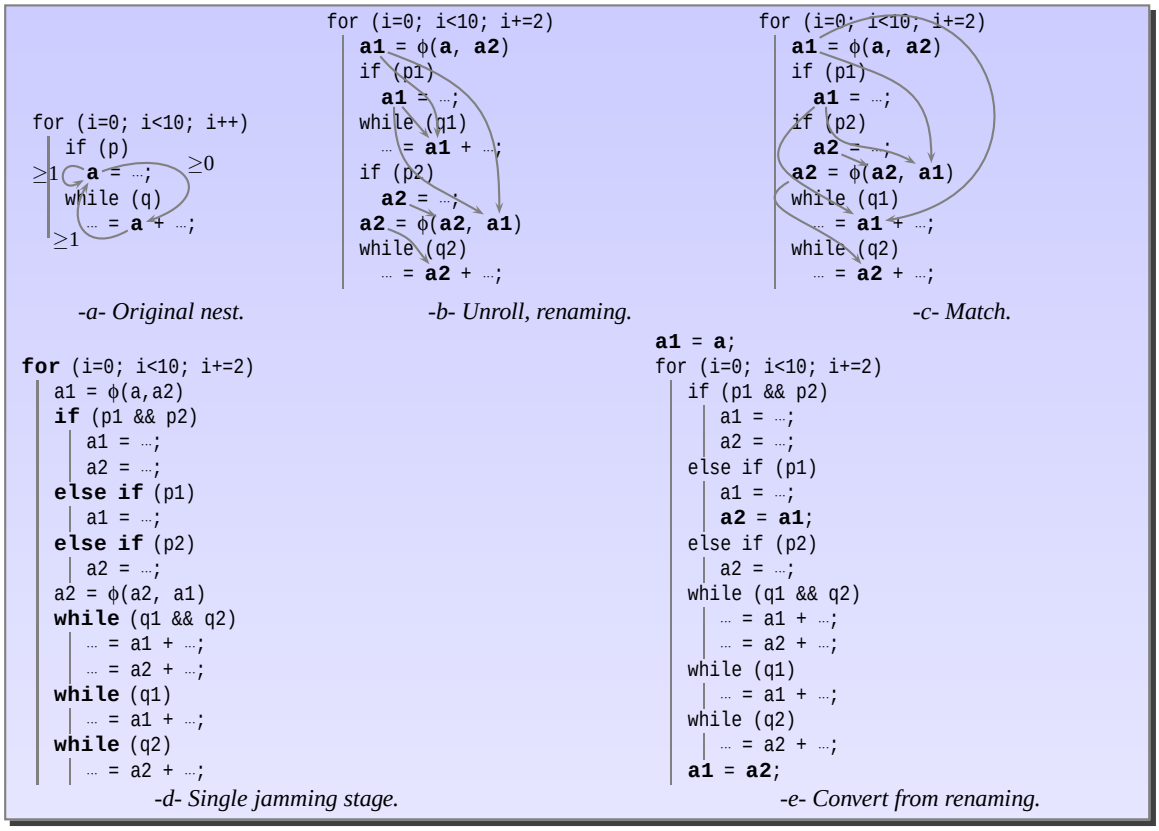


Figure 1: Irregular jam with scalar dependences.

2.1. A Single Jamming Stage

Throughout the deep jam process, the term *threadlet* will name any structured code fragment candidate for jamming with another one. In the previous example (Figure 1), each `if` structure and `while` loop represent a threadlet.

A stage of the jamming process uses the control-dependence graph (CDG) [24] to extract and process the threadlets. Starting from any control node of this graph, applying a *single jamming stage* boils down to the following sequence of operations:

1. Among children of the parent control node, choose pairs of control structures — called *threadlets* — to be jammed together. If few matching pairs can be built this way, and if the parent control node is a loop, unroll it by an unspecified factor before identifying pairs of threadlets.[2]

2. For each pair of threadlets, apply the following operations:

   (a) Rename scalar and array variables *threadlet-wise* to remove all memory-based dependences between threadlets.

2. A factor of 2 is most of the time a good trade-off to exhibit matching threadlets without code size explosion. But this factor is not limited.

(b) Match the selected threadlets. Apply code motion on one threadlet to obtain a sequential code of these two threadlets. Assuming the lack of flow dependences across structures between the selected threadlets, this code motion is valid.

(c) Fuse each pair of loops and each pair of conditionals [21]. Generate the appropriate loop epilogs to compensate for unbalanced trip counts. Compute the cross-product of the control-state automata associated with conditionals, and generate an optimized nested conditional structure from the resulting automaton [21].

The example in Figure 1 follows this informally-stated algorithm. The parent control node is a `for` loop containing 2 children: a `while` loop and an `if` structure (Figure 1-a). Because the control flow of these 2 structures is not similar, and the root node is a loop, the first step advises to unroll the loop by a factor of 2 (Figure 1-b). Then, the algorithm selects the `if`-`if` pair of threadlets and, after applying renaming, groups it by moving the second structure next to the first one. Then, these threadlets are jammed creating a new conditional structure with the merged bodies. Finally, the algorithm goes back to step 2 and selects the `while`-`while` pair. These structures are matched and jammed according to the final steps of the algorithm (Figure 1-c). The final code corresponds to the Figure 1-d.

2.2. Threadlet-Relative Renaming

STI targets independent threads only; this is a reasonable simplification for real-time system design [19], but this would kill most jamming opportunities in our compilation context. As seen in Figure 1, we perform a threadlet-wise SSA-like renaming in order to remove memory-based dependences. This is a major improvement on [21].

Scalar Renaming. Many dependence removal techniques have been designed in the context of automatic parallelization [25, 26, 27, 28, 29]. Typically, control dependences can be converted into data dependences (if-conversion), and memory-based data-dependences (output- and anti-dependences) can be removed by expansion, like privatization or renaming. Speculation or data-flow restoration induce an execution overhead; we must make sure the extra parallelism is worth what we pay for it. Fortunately, in most cases, deep jam reschedules the program in such ways that the overhead of dependence removal techniques can be minimized. Indeed, only a renaming concerning scalars written in at least one threadlet is needed.

This *threadlet-relative* renaming is a derived form of SSA transformation [30]: variables produced before the second threadlet are subscripted by 1 while those produced inside the second threadlet and after are subscripted by 2. This partial renaming is sufficient because it removes every memory-based dependence hampering the grouping of these 2 threadlets. Phi-functions are added following the SSA rules but many of these functions are useless because they concern the same variable names. Converting from this renaming behaves like the DeSSA transformation [30].

Array Renaming. Dealing with array dependences is much more complicated. ArraySSA [28] is the most natural extension of SSA. It is a good candidate to jam irregular control structures since it does not assume any particular control-flow or dependence information, and since it is mostly an array renaming transformation.[3] Although DeArraySSA is more complex than DeSSA, the flow of

3. Unlike array expansion [25, 29] and privatization [27].

data can be regenerated with low overhead in many cases [28], although precise static analysis of the array data-flow may be required [31].

Since it is not always possible to find a DeArraySSA with low overhead, alternate solutions consist in pre-constraining array renaming to cases where it is statically known how to regenerate the correct data-flow efficiently [32, 29], based on array data-flow analysis [25, 31]. These sophisticated expansion schemes have a lower runtime overhead but require accurate static analyses.

In general, it is important to take into account the overhead of array renaming in the performance estimate of any jamming strategy. Unfortunately, few quantitative evaluations of this overhead are available for the above-mentioned expansion schemes, especially for sequential execution.

2.3. Speculative Threadlet-Relative Renaming

This *threadlet-relative* renaming does not remove every dependence: when flow dependences hamper the reordering of the threadlets candidate for jamming, a speculative variant of this renaming can be used. This is valuable when this dependence is guarded by a rarely-occurring condition. Consider the example in Figure 2-a, the parent root node contains two children with different control flows, so the loop is unrolled by a factor of 2 according to the previously-stated algorithm. The next step is the threadlet-relative renaming, detailed in the previous sections (Figure 2-b). However, despite this renaming, a dependence remains from the second `if` to the first `while`, preventing the reordering phase to match the corresponding structures. But the producer of this dependence is guarded by `p1`: by speculating that this `if` is not taken, the reordering can be done, but a recover mechanism is necessary if the speculation was not appropriate.

Let t_1 and t_2 be two threadlets that cannot be jammed because of flow dependences. Assuming that a dependence analysis gives us the set S containing every block producing such dependences, every block in this set will then be speculated as not taken.

Generic Speculation. When no unrolling is involved, there is an intuitive way of speculating. Each time we move t_2 across a speculated block $B \in S$ – i.e. we move t_2 to the left on the CDG [24], over B –, the code of t_2 must be duplicated at the control-flow join of B to ensure that we still maintain the proper semantics of the program.

Transactional Model. If the parent control node is an unrolled loop (as in the example Figure 2), we perform a *transaction-like* transformation [33]. Each ϕ-function associated with a speculated block $B \in S$ becomes a ϕ^u_S-function, where u is the name of the *updated-state* variable associated with B. When we reach the DeSSA-like stage, the conversion of unannotated ϕ-functions does not change; only the annotated ones are influenced by the result of the speculation. Indeed, to remove the ϕ^u_S-functions, a special processing is applied: if we did not traverse B, then u is false and the ϕ^u_S-function becomes a classical ϕ-function. Otherwise, we must not commit variable updates that occurred during the misspeculated iteration – we must cancel the whole iteration. To do so, we simply do not allow variables to be updated past the first misspeculated unrolled iteration

In Figure 2-b, let us speculate that the `if` block on `p1` is not taken. Of course, this choice needs to be justified by a static or dynamic path profiling. This block is added to the set S. The ϕ-function at the control-flow merge point of these two iterations is annotated with S and u, the latter representing the variable used to guard the commit of the second iteration (Figure 2-c). This variable appears in the speculated block S: `u` is put to 1 meaning that the speculation is then incorrect. Thus the ϕ-definition `a1 =` ϕ^u_S`(a, a2)` can be read as *`a1` is equal to `a` at the entry point of the loop, and*

is updated to `a2` *if the previous iteration was correctly speculated on the set of blocks S or is not written otherwise.* Next, apply the jamming phase leads to the code in Figure 2-d.

Only annotated ϕ-functions are converted from this renaming in a special way; here the ϕ-function on `a1` is influenced by the result of the speculation. At the end of the loop, if `u` is equal to 0, then the speculation is valid, so the update of `a1` is performed. Otherwise, `a1` is not updated and the induction variable is brought back to the value it would have had at the end of the last correct iteration. In our example, when we misspeculate on the outcome of the second threadlet, we must decrement the value of the induction variable to re-execute the misspeculated iteration.[4] This leads to the code in Figure 2-e.

Annotating every ϕ-function associated to the join point of the unrolled loop is sufficient to control the data flow from the second unrolled iteration to the next one. Indeed, if a misspeculation occurs, then the second unrolled iteration has to be played again. In that case, the variable consistency is guaranteed through the non-update of every scalar written during the misspeculated iteration.

This approach has two main advantages: (1) it involves a very low overhead (almost no recovery code is needed, save for the induction variables shift) and (2) since we speculate on scarcely-taken blocks, re-runs of a single iteration rarely happen, and as such barely worsen the program performance when the speculation was wrong.

This speculative renaming can also be extended to arrays but depends on the array data-flow analysis to avoid run-time recovery overhead [34, 32].

2.4. Breaking Dependences Speculatively

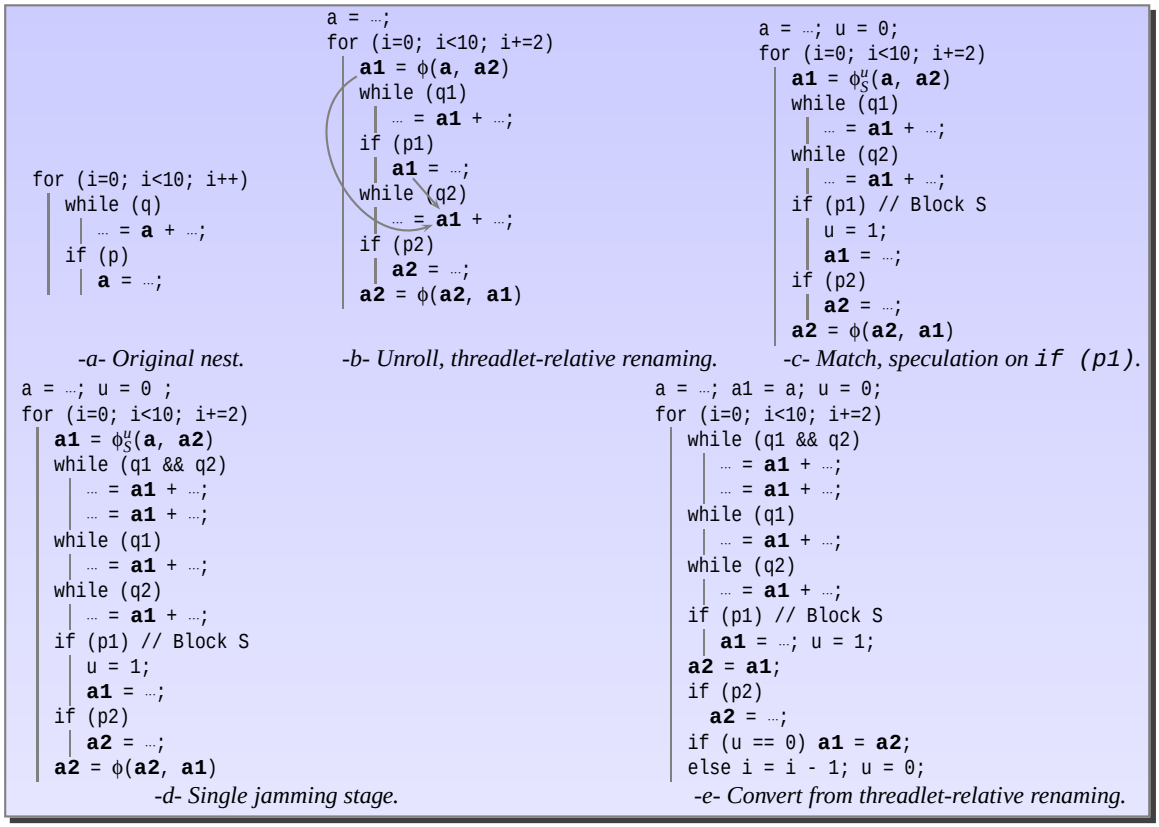
Dependences may remain after renaming (even speculative one), including def-use dependences carrying the actual flow of data and memory-based dependences whose removal through array renaming would incur too much runtime overhead. Such dependences may disappear by runtime inspection mechanisms, and more generally, any dependence can be speculatively broken with the appropriate recovery mechanism; see, e.g., [35, 36] for compile-time approaches to runtime dependence analysis and speculative parallelization. Since these techniques target massively parallel systems it is unlikely their overhead would be compatible with the comparatively limited speedup expected from deep jam.

Nevertheless, we will see in Section 5 that speculation can be profitable if restricted to critical cases where, (1) it incurs limited squash overhead, and (2) it is required to enable any jamming. In practice, it may be profitable to speculate on control-dependences due to early exits, and when ad-hoc algorithmic information can be used to avoid squashing the (whole) speculative threadlet.

2.5. Jamming Recursively

Quite naturally, jamming stages are designed to be recursively applied to inner control structures, until all the control flow dominated by the initial control statement has been covered. In addition, if an isolated loop appears at any jamming stage, it has to be unrolled by a factor of two before descending recursively in its body.[5] This way, any parallelism among outer loop iterations will

4. In general case, induction variables are updated according to the speculation outcome.
5. As an exception, innermost loops that can be efficiently optimized with traditional software pipelining (and possibly if-conversion) should not be unrolled.



-a- Original nest. *-b- Unroll, threadlet-relative renaming.* *-c- Match, speculation on `if (p1)`.*

-d- Single jamming stage. *-e- Convert from threadlet-relative renaming.*

Figure 2: Irregular jam with speculative scalar-dependence removal.

ultimately be narrowed down to inner basic blocks, hence converting coarse grain parallelism into finer grain instruction-level or vector parallelism. Of course, we are still far from an automatic deep jam algorithm. The real challenge in designing such an algorithm lies in the integration of a quantitative profitability analysis. This will be done in Section 4.

Managing code size is another challenge. Compared to STI, the urge to fuse as much control-flow as possible may lead to unacceptable expansion: special care is needed to reduce branch overhead resulting from the product control-state automaton. To mitigate this overhead, we compute — and generate code for — product-states associated with paths where basic blocks from the jammed threadlets will effectively be concatenated in further stages. Code associated with the remaining control states (single and mismatching conditional branches, `while` epilogs) is not jammed any further. For example, Figure 3 shows a binary decision tree (nested conditionals) where each node has only one (isomorphic) match when applying a jamming stage. Since jamming, e.g., a square with a circle, would not extract any additional fine-grain parallelism, the 12 associated product states are not computed, and the original subtrees are appended for default unjammed cases. This optimization preserves the amount of extracted fine-grain parallelism while avoiding the duplication of code and control for execution paths which would not benefit from the transformation.

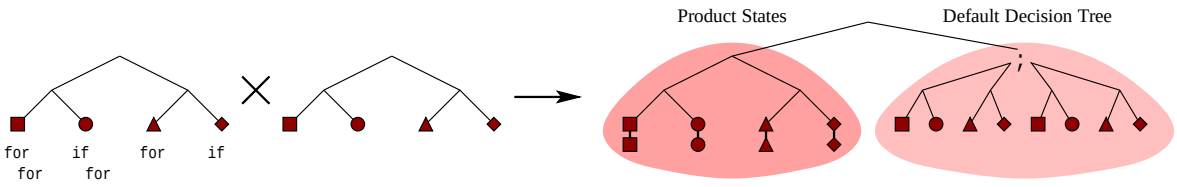


Figure 3: Jamming decision trees.

3. Jamming Criteria and Variants

The previous section outlined the primitive operations applied in our algorithm. Due to the variety of motivations, overheads and nested control structures, this algorithm needs a more complex model.

3.1. Optimization Criteria

Hot Paths. Deep jam has a chance of bringing actual speedups only when significant parts of the execution trace traverse jammed control paths: a single jamming stage should consider *all* pairs of matching control structures to maximize opportunities of building larger basic blocks from independent threadlets. From feedback-directed optimization, one may promote the formation of larger basic blocks occurring on hot execution paths. To reduce control overhead and lower register pressure, one should not fuse basic blocks occurring on cold paths in general. However, special cases exist when jamming results in simplified control-flow (factoring identical conditions) or when hot inner loops cannot be jammed without processing colder enclosing control structures: fusing control-structures enclosing the hot basic-block may require prior fusion of external colder one. For example, a cold `if` conditional including a hot `while` loop may have to be jammed to exhibit the potential gain in fusing the inner `while` loops.

Trip Count. Dynamic information is needed to make the optimization profitable. Loop jamming depends on the loop trip count and on its stability. Indeed, jamming `while` loops in the previous example will be efficient if the respective trip counts are close. Furthermore, when jamming loops whose trip-count is often close to zero, it is critical to make sure that no additional branches will be encountered on short execution paths (e.g., on zero-trip cases), compared to the original non-jammed loops.

Impact on ILP. Besides profile information, feedback from the effectiveness of a jamming strategy is needed to quantify its benefit on ILP — through schedule or vectorization improvements. If deep jam is used in an iterative optimization environment [37, 38], we may assume instruction-per-cycle (IPC) statistics are available for each basic block and for each variant; such statistics can be easily obtained from actual runs and hardware counters, or using static estimates [39]. These measurements take into account transformations applied in the back-end part of the compiler, these transformations having a strong impact on the profitability of our technique [40].

Compiler Transformations. From the compiler/architecture features and dynamic feedback, the way to jam two threadlets may evolve and lead to different possible variants. This is due to specific architectural features or compiler potential transformations.

If-conversion is an important optimization (and enabling transformation) on architectures which support predicated execution (like Intel Itanium or Philips TriMedia). The profitability heuristic for converting conditionals to predicated instructions needs to be revised in our context, for two reasons:

- predication reduces code duplication from the product-state control automaton;
- deep jam is profitable when coalescing basic blocks allows to feed idle functional units: this effect will be reinforced in a predicated implementation.

Practically, we found that if-conversion was more profitable than usual when applied to nested conditionals, and to sequences of conditionals that were not fused by deep jam.

Interestingly, if-conversion can also improve the performance of jammed loops: if an execution profile shows that the trip-count difference is much lower than the total number of iterations, it is advisable to speculatively let the shorter loop continue until the termination of the longest one, predicating loop bodies accordingly.

Tail-duplication is often associated with if-conversion to improve software pipelining [8]. Deep jam has a similar impact on the tail-duplication heuristic as on if-conversion: if a significant part of the execution is spent on non-fused code (after jamming all matching control structures), tail-duplication can enable further jamming, e.g., of loop epilogs with subsequent straight-line code from independent threadlets.

Eventually, as unroll-and-jam is not limited to unrolling factors of 2, it is possible to extend deep jam to triples of matching control structures, or even more. Our current experience shows that the control overhead and code size increase practically offset the additional ILP extraction. But this extension should be considered on wider-issue architectures like grid processors [41, 42].

3.2. Jamming Variants and Quantitative Evaluation

We first model the variants and profitability of the jamming of *leaf* control structures: innermost control nodes enclosing straight-line code, then extend it to *nested* structures.

3.2.1. Jamming Leaf Control Nodes

We detail the different possibilities of jamming single control structures with their relative quantitative evaluation. Indeed, each jamming variant may be evaluated with respect to a set of characteristic parameters of the application and architecture. In this paper, we will focus on three specific frequently-used pairs of threadlets: `if-if`, `while-while` and `for-for`.

Along these evaluations, W denotes the issue width of the processor (e.g. 6 on the Itanium 2) and P its branch misprediction penalty.[6] Each jamming variant of a pair of threadlets is statically evaluated and its evaluation is denoted $\mathrm{NC}_{t,v}$: the number of cycles to execute the jammed code of t-type threadlets with the variant v. Furthermore, we suppose that static and/or feedback-directed analyses have gathered the following set of parameters: (1) IPC_1 (resp. IPC_2) the average number of instructions per cycle for the first (resp. second) threadlet, (2) i_1 (resp. i_2) the number of instructions for the first (resp. second) threadlet, after back-end optimization (3) n_1 (resp. n_2) the number of iterations if the threadlet involves a loop and (4) $i_{1\&2}$, $\mathrm{IPC}_{1\&2}$ and $n_{1\&2}$ the corresponding metrics for the jammed body of these two threadlets.

6. Such parameters are rough estimates in general, but they happen to be quite effective on the IA64 architecture.

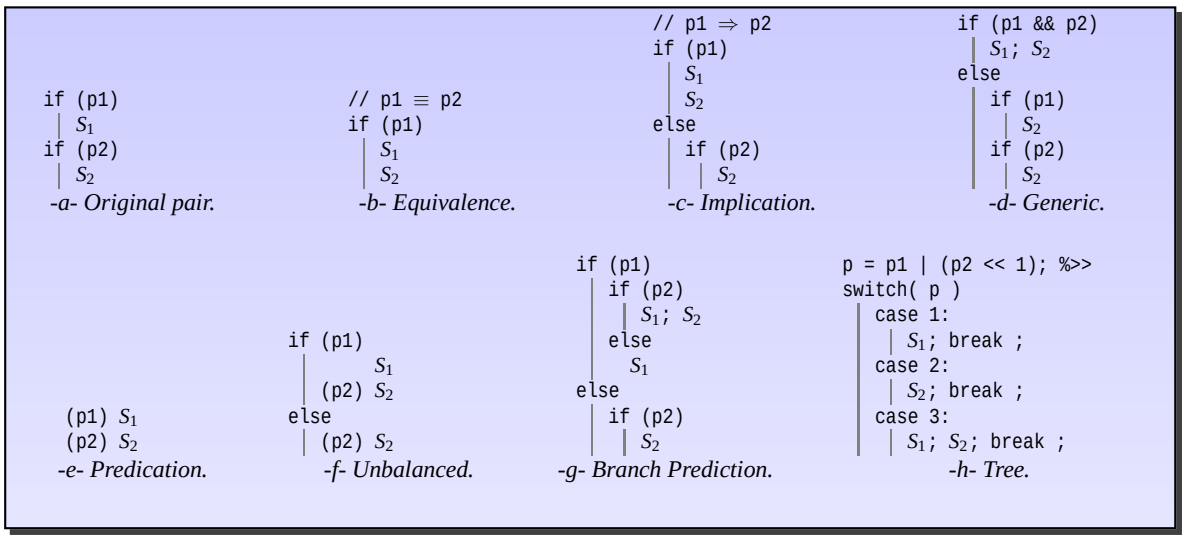


Figure 4: Jamming variants of `if`-`if` threadlets.

Jamming Variants for `if`-`if` Threadlets. Figure 4 depicts the different ways to jam two matched `if` structures, and Table 1 its associated jamming variant evaluations. The original sequential code is presented in Figure 4-a.

A static data-flow analysis may exhibit correlations between the conditions `p1` and `p2` leading to the possible jammings in Figure 4-b and Figure 4-c. Notice these cases may occur resulting from the unrolling of a surrounding loop.

When static analysis fails, the most generic form of jamming is presented in Figure 4-d. Furthermore, if the target architecture handles predication, then the variant in Figure 4-e is viable only if the two `if`s are mostly taken or when their bodies do not reach a significant size. Indeed, the size of S_1 and S_2 plays an important role in the jamming decision: if these two blocks are unbalanced, the smallest one or the most taken one can be integrated inside the other to reduce the control overhead. This leads to the variant in Figure 4-f.

Branch predictors are accurate as soon as a set of branches exhibits a bias in their respective outcome. Of course, if the threadlets are not taken — when both `p1` and `p2` are false most of the time — then it is useless to jam, but if this case occurs during the execution alternatively with a situation beneficial for jamming, then the algorithm has to deal with the potential branch mispredictions as the variant in Figure 4-g. When both conditions are verified, then no misprediction occurs (assuming `then` branches are predicted *taken*), and, when both predicates are false, then, at most, only 2 mispredictions arise compared to 3 with previous variants. The final variant requires a robust code generator because it results in a decision tree: Figure 4-h. Each condition represents a specific bit-matching pattern of `p` and then, a simple `switch` on its value determines which condition was verified or not.

Jamming Variants for `while`-`while` Threadlets. Consider now a pair of threadlets containing a `while` loop as shown in Figure 5-a. An estimate of the number of cycles spent in that unjammed pair of loops can be found in Table 1.

A performance estimate for the *pessimistic* strategy depicted in Figure 5-b is shown also in Table 1. The "+1" in the instruction count stands for the computation of the conjunction of the loop

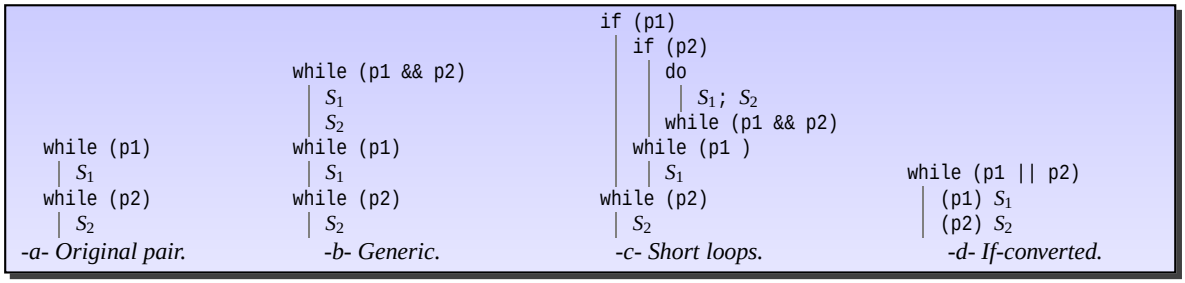


Figure 5: Jamming variants of `while-while` threadlets.

conditions. Notice $\text{IPC}_{1\&2}$ may be over-approximated by $\min(\text{IPC}_1 + \text{IPC}_2, W)$, which corresponds to an ideal interleaving of instructions from both threadlets.

If the trip count of at least one loop is low, then the variant in Figure 5-c can be considered. Indeed, at least one misprediction is saved with respect to the *pessimistic* case, and in the best case, the branch predictor may learn the behavior of the outer conditional, saving up to two mispredictions. There is a benefit on short loops only: the extra control complexity and code size may degrade the applicability of back-end optimizations and instruction cache performance.

Conversely, an *optimistic* strategy in Figure 5-d bails out when both conditions are invalidated, predicating the execution of Let $n_{1|2}$ denotes the average number of iterations of the fused part; a performance estimate for the *optimistic* strategy is shown in table 1.

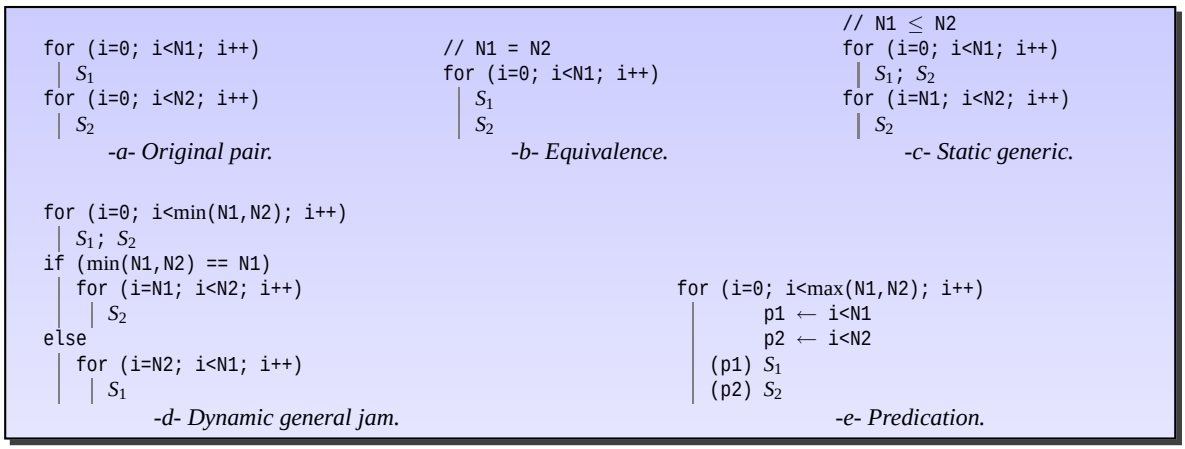


Figure 6: Jamming variants of `for-for` threadlets.

Jamming Variants for `for-for` Threadlets. Finally, we deal with the jamming of two `for` threadlets presented in Figure 6-a. In Table 1 the number of iterations of the first loop is denoted N_1 and the second one N_2 because the number of iterations is static. In the first formula, the number of branch mispredictions x can be equal to 0, 1 or 2 but, on modern architectures, counted loops prevent a misprediction. Therefore, we consider no misprediction in further variant.

The main jamming variants depend on the ability for the compiler to compare N_1 and N_2. Indeed, if a static analysis guarantees the equality of these values or their order, the variants in Figure 6-b and in Figure 6-c can be respectively used. The variant c is relevant only when the number of iterations becomes important (otherwise, other back-end optimizations have to be turned off,

Case	Evaluation	Number of branch mispredictions x
$\mathrm{NC}_{\texttt{if},a}$	$\frac{i_1}{\mathrm{IPC}_1} + \frac{i_2}{\mathrm{IPC}_2} + xP$	$\{0,1,2\}$
$\mathrm{NC}_{\texttt{if},b}$	$\frac{i_{1\&2}}{\mathrm{IPC}_{1\&2}} + xP$	$\{0,1\}$
$\mathrm{NC}_{\texttt{if},c}$, $\mathrm{NC}_{\texttt{if},d}$	$\frac{i_{1\&2}}{\mathrm{IPC}_{1\&2}} + xP$	$\{0,1,2\}$
$\mathrm{NC}_{\texttt{if},e}$	$\frac{i_{1\&2}}{\mathrm{IPC}_{1\&2}}$	
$\mathrm{NC}_{\texttt{while},a}$	$\frac{n_1 i_1}{\mathrm{IPC}_1} + \frac{n_2 i_2}{\mathrm{IPC}_2} + 2P$	
$\mathrm{NC}_{\texttt{while},b}$	$\frac{n_{1\&2}(i_1+i_2+1)}{\mathrm{IPC}_{1\&2}} + \frac{(n_1-n_{1\&2})i_1}{\mathrm{IPC}_1} + \frac{(n_2-n_{1\&2})i_2}{\mathrm{IPC}_2} + 3P$	
$\mathrm{NC}_{\texttt{while},c}$	$\mathrm{NC}_{\texttt{while},b} - 2P$	
$\mathrm{NC}_{\texttt{while},d}$	$\frac{n_{1\|2}(i_1+i_2+1)}{\mathrm{IPC}_{1\|2}} + P$	
$\mathrm{NC}_{\texttt{for},a}$	$\frac{N_1 i_1}{\mathrm{IPC}_1} + \frac{N_2 i_2}{\mathrm{IPC}_2} + xP$	$\{0,1,2\}$
$\mathrm{NC}_{\texttt{for},b}$	$\frac{N_1 i_{1\&2}}{\mathrm{IPC}_{1\&2}}$	
$\mathrm{NC}_{\texttt{for},c}$	$\frac{N_1 i_{1\&2}}{\mathrm{IPC}_{1\&2}}$	
$\mathrm{NC}_{\texttt{for},e}$	$\frac{\max(N1,N2)(i_{1\|2}+2)}{\mathrm{IPC}_{1\|2}}$	

Table 1: Jamming variant evaluation of `if-if`, `while-while`, and `for-for` threadlets (see Figures 4, 5 and 6).

like prefetching or software pipelining) and when the difference between the two trip counts is significant too. Otherwise, the *tail* code has to be generated in another way (software pipelining, versioning, ...). Finally, if the compiler and/or the target architecture handle if-conversion and the trip count of both loops is close, then the loop with the lowest trip count can be aligned on the largest by predicating the corresponding blocks. The resulting code is presented in Figure 6-e. The number of instructions of this new loop body is $i_{1|2}+2$ — to model the computation of each predicate — and an IPC equal to $\mathrm{IPC}_{1|2}$.

3.2.2. Jamming Intermediate Control Nodes

Non-leaf structures with nested control may immediately benefit from a jamming stage, if they contain significant straight-line blocks with chains of dependent instructions. More generally, the profitability of jamming intermediate control nodes derives from the further jamming stages they enable on nested control structures. One may adapt the previous performance estimates to handle this case, thanks to two simple observations:

1. instruction counts i_1 and i_2 correspond to the number of dynamically executed instructions in every inner conditional structure and block of straight-line code;

2. the IPC for each version can be derived from the division of the previous instruction count by the sum of the performance estimates of the same inner structures.

4. Deep Jam Algorithm

Deep jam is more complex than a recursive application of the single jamming stage defined in Section 2.1.. A wide spectrum of transformations, static analyses and performance estimations must be coordinated in a complex interplay. Selecting a profitable strategy within the resulting search space seems challenging.

The deep jam algorithm starts from any node in the CDG and takes 3 steps. First of all, every jamming variant among pairs of threadlets inside the current body is generated. Then, for each previously generated tree, its performance is evaluated — statically or dynamically — before choosing the best one.

Variant Generation. Figure 7 summarizes the algorithm generating variants of a code after application of deep jam. It tries iteratively to jam all matching pairs of threadlets, considering all possible variants in a breadth-first fashion. A queue $\mathcal{F}$ stores a tuple of 3 elements: the generated tree, its associated current node and a list of threadlet pairs to test. Initially, it contains CDG t, the root node r and an empty set. As the list is initially empty, the next iteration has to look for among child nodes. Thus, the algorithm finds all possible matching pairs of threadlets among children of the current node. If current node is a loop it also considers unrolling the loop by a factor of two (or more) to form new threadlets. The potential updated tree with the same root and the list L is then appended to $\mathcal{F}$.

When the element retrieved from $\mathcal{F}$ contains a non-empty list L, then its first element (c, c') indicates two subtrees candidate for jamming. This pair is then dequeued and processed: if these threadlets can be jammed together, guided by the call of *possibleJam*, then each variant is built thanks to the module *buildAllVariants* and stored inside the set V. The new tree is generated according to each variant and appended to the queue with the updated threadlet-pair list. Finally, the whole set of jammed codes T is returned.

Two functions determine the scope and efficiency of the algorithm: *possibleJam* and *buildAllVariants*. The first one checks if a pair of threadlets can be reordered in order to be matched together (thanks to threadlet-relative renaming with or without speculation — see Sections 2.2. and 2.3.). Notice this module has to store the set of speculated blocks S in order to avoid the useless jamming of speculated blocks. The second function is *buildAllVariants* which iterates over the variants proposed in Section 3.2. depending on threadlet type, content, static and/or dynamic feedback.

Profitability Evaluation and Selection. The second step executes or estimates the IPC of each code in T. (Section 3.1..) With these measurements, an inner-to-outer profitability analysis is run. (Section 3.2..) Finally, the code with the highest profitability is chosen.

The output of the whole algorithm is a jammed code with the best potential profitability. The first step (see Figure 7) is realistic only if the number of control nodes is quite small (size of t). In practice, the depth of an exhaustive search for the best jamming strategy should be bounded.

Towards an Implementation. Fortunately, the manual application of deep jam in the following section tends to indicate that the size of the search space is reasonable. Indeed, only a few alternative schemes compete for each jamming operation. Due to the nature of the quantitative performance estimates, a practical algorithm should combine static information and dynamic feedback (application profile and iterative optimization runs) [43, 44, 38].

Although we did not yet implement deep jam in a compiler, the previous study and algorithm allow us to outline a more practical deep jam algorithm refining the one presented in Figure 7:

Inputs: t Control-dependence graph
r Root of the control-dependence graph
Output: T Set of variants
Algorithm (Deep jam's variant generation):

$T \leftarrow \emptyset$; $\mathcal{F} \leftarrow (t, root, \emptyset)$
While $\mathcal{F} \neq \emptyset$
$(t', r', L) \leftarrow nextElement(\mathcal{F})$
$\mathcal{F} \leftarrow \mathcal{F} \setminus (t', r', L)$
If $L = \emptyset$
$L' \leftarrow findAllMatchingPairs(r', t')$
If $L' = \emptyset$ and r' is a loop
$t'' \leftarrow unroll(t', 2)$
$L'' \leftarrow findAllMatchingPairs(r', t'')$
$\mathcal{F} \leftarrow \mathcal{F} \cup (t'', r', L'')$
Else
$\mathcal{F} \leftarrow \mathcal{F} \cup (t', r', L')$
Else
$(c, c') \leftarrow firstElement(L)$; $L \leftarrow L \setminus (c, c')$
If $possibleJam(t', c, c')$
$V \leftarrow buildAllVariants(t', c, c')$
For $v \in V$
$t'' \leftarrow generateVariant(t', v, c, c')$
$T \leftarrow T \cup t''$
If $L = \emptyset$
$r'' \leftarrow nextBreadthFirstElement(r', t')$
$\mathcal{F} \leftarrow \mathcal{F} \cup \{(t'', r'', \emptyset)\}$
Else
$\mathcal{F} \leftarrow \mathcal{F} \cup \{(t'', r', L)\}$

Return T

Figure 7: Deep jam — variant generation algorithm.

1. Starting from the parent control node, unroll it by a factor of 2 if this is a loop (the probability that each loop initially exhibits similar control flow is not significant). In our experiments, the parent node has always been a loop containing the most time consuming part of the procedure. If several loops share the time spent in this procedure, they can be processed separately.

2. Apply threadlet-relative renaming on the two loop bodies to remove superfluous dependences flowing from one iteration to the other. Potentially, apply speculative renaming if the set of speculative blocks is small and located on cold paths.

3. Among the children of the loop, try the different jamming variants for each pair of matching threadlets.

4. Repeat recursively the previous step for each pair of threadlets. The level of recursion has to be bounded by 2 or 3.

These steps simplify the previously-stated algorithm by unrolling the outer loop, renaming only once and recursively jamming each matching pair. But the level of recursion is bounded (typically by 2 or 3) to reduce the parameter space.

Overall, even if, thanks to the previous guidelines, the compilation time expected is reduced, deep jam is a challenging compilation problem. It involves complex transformations, relies on precise static analysis, including array-dependence analysis, and its profitability is hard to assess statically. In addition, although deep jam combines multiple (classical and original) transformations, applying any of these transformations in isolation does not bring any speedup or may even degrade performance. Nevertheless, our experiments in the next section will confirm that it can bring strong speedups. Moreover, it will show that, with precise static cost models, the compilation time could be low enough to incorporate deep jam inside a compiler.

5. Experiments

Let us study four real compute-intensive applications characterized by long sequences of dependent instructions, irregular control-flow and intricate scalar and array dependence patterns. We apply deep jam at programmer level as source-to-source transformation using the Intel ICC compiler to generate the binary object. We followed the guidelines of the practical algorithm. Special care was taken to ensure that the compiler did not undo transformations made by deep jam. The following experiments demonstrate the strong potential of deep jam, exercising the tuning of the main parameters driving the selection of a profitable deep jam strategy.

5.1. SHA-0 Attack

We first study the attack of the SHA-0 cryptographic hash algorithm [45], which lead to a full collision in August 2004 [46, 47]. This algorithm belongs to the family of iterative hash functions. It relies on a compression function f taking as input a message and a tuple of five 32-bit values. The application of f returns another tuple forming, after an addition with the initial one, the 160-bit hash value of the message. Compression is decomposed into 80 *rounds* of (mainly) bitwise operations. The attack applies the SHA-0 algorithm iteratively to a pair of messages, checking at each round if they may possibly collide or not at the end (i.e., after the 80 rounds). The research of colliding messages is not exhaustive: messages are tested so that first computations (more or less the first 14 rounds) can be reused from a pair of messages to another, leading to a rather irregular control structure with guarded compute kernels and early exits.

The experimental platform is a NovaScale 4020 server from Bull featuring two Itanium 2 1.3GHz (Madison) processors, using the Intel C compiler version 8.1, choosing the best result from `-O2` and `-O3` with `-fno-alias`.

Performance analysis of this code highlights several limiting factors: memory pressure, complex control flow and limited amount of parallelism. To release memory pressure, we apply two optimizations: *scalar promotion* (via loop unrolling), then *vectorization* of straight-line 32-bit operations (using 64-bit registers and SIMD instructions), to save registers and avoid the spills created by the previous step (and of course, to reduce the number of operations). Strangely, this version

does not provide significant speedup. Hardware counters detect a large rate of pipeline stalls due to register-register dependences:

IPC = 2.48 `nops` = 13.4% Reg-reg stalls = 15.0%

This code is composed of a main loop iterating on messages. Because, at this level, this loop is alone and children of this control node contains no clear matching pairs, the deep jam algorithm first unrolls this loop by a factor of two. The loop body is large (a thousand lines of code, implementing up to 80 rounds on the selected message), and its control flow is apparently unpredictable. Alone, this transformation only brings 1% speedup.

Before attempting to jam resulting threadlets (instances of every inner conditional and loop in the unrolled body), a large number of scalar dependences are eliminated by threadlet-relative renaming. One array of 80 elements needs to be renamed to remove output and anti-dependences. After this expansion step, the remaining def-use dependences are compatible with a one-to-one fusion of every matching pair of conditionals and inner loops.

Yet several control-dependences remain; they are due to early exits in the acyclic part and in the single inner `for` loop. Speculatively ignoring these dependences degrades performance, and tail-duplication is not applicable because of data-sensitive predicates guarding control-dependences. As a result, some control-dependent code cannot be jammed as effectively as expected. For example, the inner loop is jammed with its matching pair using the *pessimistic* strategy in Figure 5-b, instead of an optimized scheme with if-conversion. Feedback from a dynamic profile tells that the first three rounds are only sparsely executed, hence the associated `if` conditionals do not need to be jammed; this saves the generation of a 9-case decision tree and reduces code size.

The resulting code is approximately 4 times larger than the original application (due to unrolling and `while` loop epilogs), and provides a **43.3%** speedup. Hardware counters reveal a major improvement on the number of stalls and `nops`:

IPC = 3.17 `nops` = 10.3% Reg-reg stalls = 7.71%

5.2. ABNDM/BPM String Matching

The second application optimized by deep jam comes from computational biology. It implements an approximate pattern matching algorithm, named ABNDM/BPM [48], which finds all positions where a given pattern of m characters matches a text with up to k differences (substitution, deletion or insertion of a character).[7] Assuming an online search, the pattern is known and can be pre-processed to speedup the search, but the text may not. ABNDM/BPM is a key contribution to the pattern matching domain, since it combines dynamic programming, filtering and bit-parallelism [49]. The text is processed through windows of $m-k$ characters, to decide if an occurrence may appear inside a window and how many characters to skip (less than $m-2k$) before the next window. Approximate matches are selected from the bit-parallel simulation of a non-deterministic finite-state automaton with a dynamic programming matrix [50, 49].

The code is composed of a main loop, iterating on the text, window after window. The loop body contains early exists, conditionals and nested while loops. The processing of a window is split into a first phase, traversing the window backwards. A first `for` loop iterates unconditionally on k characters, then a `while` loop proceeds with at most $m-2k$ iterations. The skip distance between

7. In practical searches, k can be as large as $m/2$.

two consecutive windows is computed dynamically as a result of this backward phase, depending on the text being traversed. If the `while` loop effectively completed the traversal (reading all characters in the window), a second phase traverses the window forward, checking if an occurrence appears (beginning at the first character).

The experimental platform is a 800MHz Itanium (Merced) 4-way SMP, using the Intel C compiler version 7.1, choosing the best result from `-O2` and `-O3` with `-fno-alias`.

Again, the analysis of the generated assembly code and hardware counters indicate a lack of ILP in chains of dependent instructions. In addition, the complex data-dependent control is reflected in the high rate of pipeline flushes: up to 30% of the execution time is waisted in mispredicted branches. For typical cases, the IPC lies between 1.3 and 1.5.

Deep jam is only applied on the backward phase, since it amounts to more than 90% of the computation time. Because the main loop has no candidate for jamming, it first unrolls this loop, yielding several threadlets associated with the backward traversal of two subsequent windows. The control flow of this backward phase is quite complex and dependent on the input data. The unrolling transformation alone does not bring any speedup.

Unfortunately, one immediately notices that the dynamic computation of the skip between two consecutive windows yields several control and data dependences. Indeed, even if a large number of scalar dependences are eliminated by threadlet-relative renaming, any jamming scheme needs to speculatively break those dependences. Thanks to domain-specific knowledge, we know that under-approximating this distance is a conservative solution (yielding lower performances but still covering all possible matches). Figure 8 sketches a speculative jamming scheme where the position of the second window is estimated at each iteration of the outer (unrolled) loop.

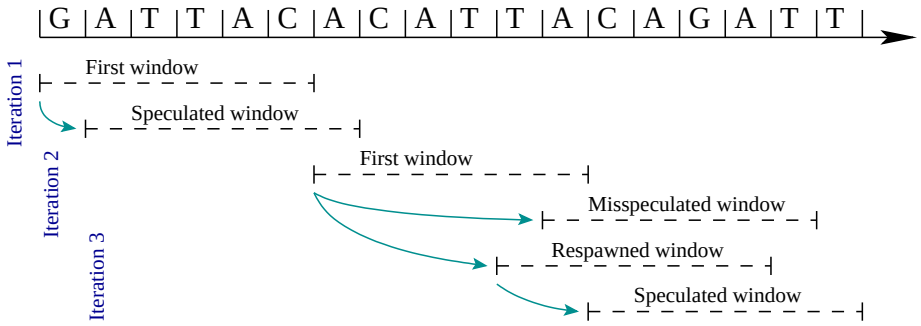


Figure 8: Jamming windows speculatively.

The next difficulty comes from the jamming of a very short inner `while` loop nested in a complex decision tree. This section is responsible for most branch mispredictions identified in the preliminary analysis. Interestingly, the *optimistic* jamming strategy of Figure 5-d results in a strong reduction of the mispredictions rate, through tail-duplication and if-conversion. This strategy will be called `optim` in the following experiments. However, the reduction in mispredictions is not always beneficial, due to the unnecessary (predicated) work overhead in the frequent cases where the inner `while` loop executes less than 3 iterations. We will thus also consider a *short loop* strategy, called `pessim` thereafter, as defined in Figure 5-c. For some input text and values of k, the length of the backward window traversals is very unstable. This reduces deep jam benefits, since most of the time will be spent in unjammed loop epilogs. It may be more effective to squash the execution

of the second window when the first one terminates early, and restart the traversal from the beginning (at a non-speculative position), jammed with the *subsequent* backward window traversal. This strategy, called priority thereafter, also simplifies the control-flow, eliminating complex loop epilogs. This strategy is not easily generalized to other deep jam cases, hence its absence from the jamming variants of Section 3.1..

Figure 9 shows the speedups achieved on the full application, varying the input text and the number of errors k, with fixed pattern size $m = 32$. Since no jamming strategy dominates in all contexts, all three are evaluated. The best speedup reaches **58.9%**, but using the wrong strategy leads to significant slowdowns. We thus designed an adaptive selection scheme, to dynamically select the best strategy. We observed that the priority strategy is not profitable if the rate of early exits is high, i.e., if the backward phase quickly discovers that no match is possible. The adaptive scheme thus begins in the priority mode, then switches to the optim strategy if the number of early exits reaches a certain threshold. This scheme is fully automatic and incurs only 1% performance degradation compared to the best speedup achieved with either priority or optim. This adaptive selection could be extended to the pessim scheme, based on an instrumentation of inner loop trip-count; yet the benefits would be moderate since pessim rarely dominates.

k	priority	pessim	optim
0	-6.5%	-73.6%	58.9%
1	-7.9%	-68.7%	56.5%
2	-8.2%	-63.2%	49.9%
3	-9.7%	-56.7%	46.5%
4	-12.4%	-48.7%	41.5%
5	32.3%	30.5%	22.5%
6	33.1%	30.9%	22.5%
7	30.4%	29.2%	21.2%
8	28.2%	27.4%	19.8%
9	13.3%	-13%	53.2%
10	13.6%	7.8%	57.2%

-a- English dictionary.

k	priority	pessim	optim
0	39.2%	38.2%	26%
1	37.7%	34.2%	24.4%
2	35.5%	35.7%	22.3%
3	33.4%	31.1%	25.2%
4	-11.8%	-40%	26.5%
5	-12.8%	-31%	24%
6	-16.5%	-24.6%	21.8%
7	-13.4%	-16%	18.9%
8	-17%	-17.6%	20.6%
9	-14.7%	7.4%	39.4%
10	11.8%	10.6%	6.3%

-b- LaTeX document.

k	priority	pessim	optim
0	-11.5%	-68.1%	33.7%
1	-18.7%	-58.4%	20.5%
2	-18.2%	-46%	21.4%
3	-19%	-33.9%	21.8%
4	-21.3%	-27.5%	17.4%
5	-20.9%	-22.1%	15%
6	-21.4%	-18.7%	11.9%
7	-21.3%	-17.2%	9.4%
8	-22.5%	-16.8%	7%
9	-22.4%	-16.3%	4.2%
10	-22.4%	-15.5%	2%

-c- DNA of Buchnera bacterium.

k	priority	pessim	optim
0	-15.4%	-69.8%	32.6%
1	-22.1%	-59.3%	18.0%
2	-20.3%	-48.9%	23.2%
3	-20%	-36.3%	26.1%
4	-21.8%	-31.0%	19.7%
5	-21.7%	-25.2%	18.5%
6	-20.7%	-18.6%	16.2%
7	-21.3%	-16.4%	12.5%
8	-22.5%	-14.8%	9.1%
9	-22.4%	-14.2%	4.6%
10	-23%	-14.2%	3.4%

-d- DNA of bacillus anthracis str. Ames.

Figure 9: Performance of three deep jam variants on ABNDM/BPM.

	CPU Cycles (whole application)	IPC	Speedup
Original version	373.174×10^9	2.51	—
Unroll (factor of 2)	372.919×10^9	2.43	0.1%
Deep jam	350.809×10^9	2.51	6.4%

Table 2: BLAST — Optimization of the most time consuming loop.

5.3. BLAST

The next benchmark also belongs to computational biology: BLAST (Basic Local Alignment Search Tool)[8] is a set of algorithms used to find similar sequences between several DNA chains or protein databases. Unlike the previous algorithm, BLAST only finds *exact matches*. This program, composed of dozens of functions, is an integer, computational-intensive code exhibiting an important control-flow complexity.

The most time consuming loop is a `for` loop including a dominating `if` conditional with an unbalanced `else` branch. A quick dynamic analysis confirms ILP is low on the most frequently used path. Indeed, the most executed path is only the first basic block of the loop body, followed by the short `then` branch of the `if` conditional. The first basic block loads fields from an array of records, just before operating on them. This leads to short dependence chains, reducing ILP even if the compiler assumes that data are prefetched into the L1 cache (latency of 1 cycle).

Because the `for` does not have potential matching threadlets among its children, the first step consists of unrolling this loop by a factor of 2. After the threalet-relative renaming, the first basic block of each iteration is matched and jammed (jamming basic blocks is straightforward). But this can be done only by speculating on the value of `diag_coord`. Indeed, in the `then`, the `diag_coord` cell of array `diag_array` is written. During the next iteration, the same array is read. If the indices match, the code motion is not valid. Therefore we matched and jammed the first instructions of the loop by checking the outcome of the speculation after the `if-else` structure of the first iteration, applying a speculative threadlet-relative renaming to `diag_coord`.

We did not jam further because, (1) path profiling indicates the `then` branch is a hot-path but the body is very short and (2) the speculation induces control overhead leading the a higher complexity hampering further jamming. So the final code features a loop unrolled twice, a match of the first basic block of each iteration and, finally, a speculation on `diag_coord`.

Table 2 shows results obtained on a 1.6 GHz Itanium 2 Madison machine with 3MB of L3 cache and ICC Compiler version 9.1. Three versions have been tested: the original code, the original code unrolled by a factor of 2 and, finally, the code after applying deep jam.

This table presents, for each version, the time spent in the *whole application* plus the corresponding IPC and the speedup w.r.t. the original code. The IPC of the whole application is good for an integer irregular code. The unrolling does not bring any speedup. Indeed, after unrolling, code motion is impossible due to the potential dependence on `diag_array` through the value of `diag_coord`. The deep-jammed code brings more than 6% speedup on the whole application and keep the IPC at the same level. Actually, its performance come from the merge of the basic blocks at the beginning of the `for` loop feeding functional units and recovering load latencies.

8. See `http://www.ncbi.nlm.nih.gov/BLAST/`

Input set `graphic`	CPU Cycles	IPC	% Bubbles	Speedup
Initial	13.724×10^9	1.33	44.2%	—
Unroll (factor of 2)	12.824×10^9	1.45	46%	7%
Deep jam	11.842×10^9	1.73	44.9%	15.9%
Input set `log`	CPU Cycles	IPC	% Bubbles	Speedup
Initial	12.797×10^9	1.70	34.2%	—
Unroll (factor of 2)	11.299×10^9	1.86	36.2%	13.3%
Deep jam	10.464×10^9	2.33	38.8%	22.3%
Input set `program`	CPU Cycles	IPC	% Bubbles	Speedup
Initial	102.748×10^9	1.67	22.3%	—
Unroll (factor of 2)	84.087×10^9	1.86	26.8%	22.2%
Deep jam	73.186×10^9	2.62	28.8%	40.4%
Input set `random`	CPU Cycles	IPC	% Bubbles	Speedup
Initial	12.642×10^9	1.20	45.7%	—
Unroll (factor of 2)	11.846×10^9	1.34	46.7%	6.70%
Deep jam	10.903×10^9	1.60	45.8%	16%
Input set `source`	CPU Cycles	IPC	% Bubbles	Speedup
Initial	42.503×10^9	1.6	28.3%	—
Unroll (factor of 2)	35.437×10^9	1.77	31.2%	20%
Deep jam	30.629×10^9	2.33	34.4%	38.8%

Table 3: Performance results of Gzip benchmark on `longest_match` function.

5.4. Gzip

The last code optimized with deep jam is Gzip, extracted from the SPEC2000 benchmark suite. In the most time-consuming function, the outer loop iterates on the current text looking for a match. A first profiling phase highlights that the first `if` is usually taken, ending the iteration prematurely (due to the `continue` instruction).

Because this loop can not be matched with another one, the deep jam algorithm first unrolls it by a factor of 2. Even through renaming, dependences on the `scan`, `scan_end` and `scan_end1` variables remain. Indeed, the main goal is to jam two occurrences of the first `if`, but the values read during the second iteration may have been written during the first one. So the speculative threadlet-relative renaming created new scalar variables and selects the subtree after the first `if` as speculated (set *S*). Indeed, as soon as the first iteration goes through the first two `if`s and update the variable `scan` (instruction `scan += 2`), then the speculation is wrong and the second iteration must not be committed. Thus a new variable `u` is updated to 1 just before this instruction. During the commit, if `u` is equal to 1, then the induction variables `cur_match` and `chain_length` are updated from the first iteration. Otherwise, they are updated with the variables of the second one.

Experimental results are provided Table 3. The target architecture is a Madison Itanium 2 processor with 3MB of L3 cache and we use the Intel ICC compiler version 9.1. The *% Bubbles* column gives the percentage of back-end bubbles occurring in a single run, i.e. the percentage of dynamic stalls in the processor pipeline. Each table presents the results of each input set given with the Spec distribution. Overall, the IPC of each input set is low (around 1). Unrolling the loop by a factor of 2 brings a small speedup. This is due to the fact that few instructions can be moved without any further requirements. But deep jam brings more speedup, up to 40% on the `program` input set.

6. Conclusion

This papers presents a new program optimization, called deep jam, to convert coarse-grain parallelism into finer-grain instruction or vector parallelism. This optimization is applicable to irregular control and data flow where traditional optimizations fail to extract parallelism from chains of dependent instructions. It handles nested loops and unpredictable conditionals, removing memory-based dependences with modern scalar and array renaming techniques. Several experiments are conducted on a wide-issue architecture, showing that deep jam brings good speedups on real applications: 43.3% on a cryptanalysis code, up to 58.9% on computational biology applications and 40% on a SPECINT 2000 benchmark function. We detail strategies and variants associated with the jamming of irregular control structures, and integrate them in a practical deep jam algorithm. We also study the implementation of deep jam in a feedback-directed optimization framework.

Deep jam is also appealing for domain-specific program generators [51]; we believe expert knowledge (from the programmer) will dramatically smooth the implementation challenges, compared with a general-purpose compiler optimization framework. It seems interesting to evaluate deep jam for grid processors [41, 42], reconfigurable computing and hardware synthesis [52], and custom extensions to VLIW processors [53, 54]. In this context, deep jam should be extended to favor the jamming of control structures with non-conflicting usage of functional units; e.g., favoring the interleaving of floating-point and bitwise operations, or the fusion of array and scalar code [21]. Since deep jam revisits several automatic parallelization techniques for irregular programs, coupling it with hybrid static-dynamic analyses [55] may improve its effectiveness.

References

[1] J. González and A. González, "The potential of data value speculation to boost ILP," in *ACM Int. Conf. on Supercomputing (ICS'98)*, pp. 21–28, 1998.

[2] B. Calder, G. Reinman, and D. M. Tullsen, "Selective value prediction," in *Proc. Intl. Symp. on Computer Architecture (ISCA'99)*, pp. 64–74, 1999.

[3] D. Tullsen, S. Eggers, and H. Levy, "Simultaneous multithreading: Maximizing on-chip parallelism," in *Proc. Intl. Symp. on Computer Architecture (ISCA'95)*, jun 1995.

[4] A. Cristal, O. J. Santana, M. Valero, and J. F. Martínez, "Toward kilo-instruction processors," *ACM Trans. on Architecture and Code Optimization*, vol. 1, no. 4, pp. 389–417, 2004.

[5] S. E. Raasch, N. L. Binkert, and S. K. Reinhardt, "A scalable instruction queue design using dependence chains," in *Proc. Intl. Symp. on Computer Architecture (ISCA'02)*, (Anchorage, AK), May 2002.

[6] A. Moshovos, P. Banerjee, S. Hauck, and Z. A. Ye, "Chimaera: A high-performance architecture with a tightly-coupled reconfigurable functional unit," in *Proc. Intl. Symp. on Computer Architecture (ISCA'04)*, (Vancouver, BC), jun 2000.

[7] S. Yehia and O. Temam, "From sequences of dependent instructions to functions: An approach for improving performance without ILP or speculation," in *Proc. Intl. Symp. on Computer Architecture (ISCA'04)*, (Munich, Germany), June 2004.

[8] R. Allen and K. Kennedy, *Optimizing Compilers for Modern Architectures*. Morgan and Kaufman, 2002.

[9] K. Kennedy and K. McKinley, "Maximizing loop parallelism and improving data locality via loop fusion and distribution," in *Languages and Compilers for Parallel Computing*, (Portland), pp. 301–320, 1993.

[10] S. Carr, C. Ding, and P. Sweany, "Improving software pipelining with unroll-and-jam," in *Proceedings of the 29th Hawaii Intl. Conf. on System Sciences (HICSS'96) Volume 1: Software Technology and Architecture*, IEEE Computer Society, 1996.

[11] M. Lam, "Software pipelining: an effective scheduling technique for vliw machines," in *ACM Symp. on Programming Language Design and Implementation (PLDI'88)*, (Atlanta, GA), pp. 318–328, July 1988.

[12] J. Wang and G. Gao, "Pipelining-dovetailing: a transformation to enhance software pipelining for nested loops," in *Intl. Conf on Compiler Construction(CC'96)*, vol. 1060 of *LNCS*, (Linköping, Sweden), pp. 1–17, Springer-Verlag, Apr. 1996.

[13] H. Rong, Z. Tang, R. Govindarajan, A. Douillet, and G. R. Gao, "Single-dimension software-pipelining for multi-dimensional loops," in *ACM Conf. on Code Generation and Optimization (CGO'04)*, pp. 163–174, Mar. 2004.

[14] J. A. Fisher, "Trace scheduling : A technique for global microcode compaction," *IEEE Trans. on Computers*, vol. C-30, July 1981.

[15] W.-M. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery, "The superblock: an effective technique for VLIW and superscalar compilation," *J. Supercomput.*, vol. 7, no. 1-2, pp. 229–248, 1993.

[16] R. E. Hank, S. A. Mahlke, R. A. Bringmann, J. C. Gyllenhaal, and W. mei W. Hwu, "Superblock formation using static program analysis," in *MICRO 26: Proceedings of the 26th annual international symposium on Microarchitecture*, (Los Alamitos, CA, USA), pp. 247–255, IEEE Computer Society Press, 1993.

[17] S. S. Muchnick, *Advanced Compiler Design & Implementation*. Morgan Kaufmann, 1997.

[18] D. I. August, D. A. Connors, S. A. Mahlke, J. W. Sias, K. M. Crozier, B.-C. Cheng, P. R. Eaton, Q. B. Olaniran, and W.-M. Hwu, "Integrated predicated and speculative execution in the IMPACT EPIC architecture," in *Proc. Intl. Symp. on Computer Architecture (ISCA'98)*, July 1998.

[19] A. G. Dean and J. P. Shen, "Techniques for software thread integration in real-time embedded systems," in *IEEE Real-Time Systems Symposium (RTSS'98)*, (Madrid, Spain), Dec. 1998.

[20] A. G. Dean, "Software thread integration for embedded system display applications," *Trans. on Embedded Computing Sys.*, vol. 5, no. 1, pp. 116–151, 2006.

[21] W. So and A. G. Dean, "Procedure cloning and integration for converting parallelism from coarse to fine grain," in *7th Annual Workshop on Interaction between Compilers and Computer Architectures (INTERACT-7)*, 2003.

[22] W. So and A. G. Dean, "Reaching fast code faster: using modeling for efficient software thread integration on a vliw dsp," in *CASES '06: Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, (New York, NY, USA), pp. 13–23, ACM Press, 2006.

[23] P. Carribault, A. Cohen, and W. Jalby, "Deep jam: Conversion of coarse-grain parallelism to instruction-level and vector parallelism for irregular applications," in *PACT '05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques (PACT'05)*, (Washington, DC, USA), pp. 291–302, IEEE Computer Society, 2005.

[24] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Trans. Program. Lang. Syst.*, vol. 9, no. 3, pp. 319–349, 1987.

[25] P. Feautrier, "Array expansion," in *ACM Int. Conf. on Supercomputing*, (St. Malo, France), pp. 429–441, July 1988.

[26] D. E. Maydan, S. P. Amarasinghe, and M. S. Lam, "Array dataflow analysis and its use in array privatization," in *20th ACM Symp. on Principles of Programming Languages*, (Charleston, South Carolina), pp. 2–15, Jan. 1993.

[27] P. Tu and D. Padua, "Automatic array privatization," in *6th Workshop on Languages and Compilers for Parallel Computing*, no. 768 in LNCS, (Portland, Oregon), pp. 500–521, Aug. 1993.

[28] K. Knobe and V. Sarkar, "Array SSA form and its use in parallelization," in *25th ACM Symp. on Principles of Programming Languages*, (San Diego, CA), pp. 107–120, Jan. 1998.

[29] A. Cohen, *Program Analysis and Transformation: from the Polytope Model to Formal Languages*. PhD thesis, Université de Versailles, France, Dec. 1999.

[30] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Trans. on Programming Languages and Systems*, vol. 13, pp. 451–490, Oct. 1991.

[31] J.-F. Collard, "The advantages of reaching definition analyses in Array (S)SA," in *11th Workshop on Languages and Compilers for Parallel Computing*, no. 1656 in LNCS, (Chapel Hill, North Carolina), pp. 338–352, Springer-Verlag, Aug. 1998.

[32] D. Barthou, A. Cohen, and J.-F. Collard, "Maximal static expansion," in *25th ACM Symp. on Principles of Programming Languages (PoPL'98)*, (San Diego, California), pp. 98–106, Jan. 1998.

[33] M. Herlihy and J. E. B. Moss, "Transactional memory: architectural support for lock-free data structures," in *ISCA '93: Proceedings of the 20th annual international symposium on Computer architecture*, (New York, NY, USA), pp. 289–300, ACM Press, 1993.

[34] P. Feautrier, "Dataflow analysis of scalar and array references," *Int. J. of Parallel Programming*, vol. 20, pp. 23–53, Feb. 1991.

[35] M. Griebl and J.-F. Collard, "Generation of synchronous code for automatic parallelization of `while` loops," in *Euro-Par'95* (S. Haridi, K. Ali, and P. Magnusson, eds.), vol. 966 of *LNCS*, pp. 315–326, Springer-Verlag, 1995.

[36] L. Rauchwerger and D. Padua, "The LRPD test: Speculative run–time parallelization of loops with privatization and reduction parallelization," *IEEE Transactions on Parallel and Distributed Systems, Special Issue on Compilers and Languages for Parallel and Distributed Computers*, vol. 10, no. 2, pp. 160–180, 1999.

[37] T. Kisuki, P. Knijnenburg, K. Gallivan, and M. O'Boyle, "The effect of cache models on iterative compilation for combined tiling and unrolling," in *Parallel Architectures and Compilation Techniques (PACT'00)*, IEEE Computer Society, Oct. 2001.

[38] K. D. Cooper, D. Subramanian, and L. Torczon, "Adaptive optimizing compilers for the 21st century," *J. of Supercomputing*, 2002.

[39] S. Winkel, "Exploring the performance potential of itanium processors with ilp-based scheduling," in *IEEE/ACM International Symposium on Code Generation and Optimization (CGO'04)*, (Palo Alto), March 2004.

[40] W. So and A. G. Dean, "Complementing software pipelining with software thread integration," in *ACM Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'05)*, 2005.

[41] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. W. Keckler, and C. R. Moore, "Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture," in *Proc. Intl. Symp. on Computer Architecture (ISCA'03)*, (San Diego, CA), June 2003.

[42] M. B. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, J.-W. Lee, P. Johnson, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal, "The RAW microprocessor: A computational fabric for software circuits and general purpose programs," *IEEE Micro*, 2002.

[43] M. D. Smith, "Overcoming the challenges to feedback-directed optimization," in *ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization*, pp. 1–11, 2000. (Keynote Talk).

[44] T. Kisuki, P. Knijnenburg, M. O'Boyle, and H. Wijshoff, "Iterative compilation in program optimization," in *Proc. CPC'10 (Compilers for Parallel Computers)*, pp. 35–44, 2000.

[45] N. I. of Standards and Technologies, eds., *Secure Hash Standard*, vol. FIPS-180. Information Processing Standards Publication, may 1993.

[46] A. Joux, "Collisions in SHA-0." CRYPTO Rump Session, 2004.

[47] E. Biham, R. Chen, A. Joux, P. Carribault, W. Jalby, and C. Lemuet, "Collisions of SHA-0 and reduced SHA-1," in *EUROCRYPT'05*, Springer-Verlag, 2005.

[48] H. Hyyrö and G. Navarro, "Faster bit-parallel approximate string matching," in *Proc. Symp. on Combinatorial Pattern Matching (CPM'02)*, no. 2373 in LNCS, pp. 203–22, Springer-Verlag, 2002.

[49] G. Navarro and M. Raffinot, *Flexible Pattern Matching in Strings.* Cambridge University Press, 2002.

[50] G. Myers, "A fast bit-vector algorithm for approximate string matching based on dynamic programming," *J. of the ACM*, vol. 46, no. 3, pp. 395–415, 1999.

[51] J. Moura, M. Püschel, J. Dongarra, and D. Padua, eds., *Proceedings of the IEEE. Special Issue on Program Generation, Optimization, and Platform Adaptation*, vol. 93. IEEE Computer Society, Jan. 2005.

[52] B. So, M. W. Hall, and P. Diniz, "A compiler approach to design space exploration in fpga-based systems," in *ACM Symp. on Programming Language Design and Implementation (PLDI'02)*, June 2002.

[53] R. Schreiber, S. Aditya, B. Rau, V. Kathail, S. Mahlke, S. Abraham, and G. Snider, "High-level synthesis of nonprogrammable hardware accelerators," tech. rep., Hewlett-Packard, May 2000.

[54] "Silicon Hive web site." `http://www.silicon-hive.com`.

[55] S. Rus, L. Rauchwerger, and J. Hoeflinger, "Hybrid analysis: static & dynamic memory reference analysis," *Int. J. of Parallel Programming*, vol. 31, no. 4, pp. 251–283, 2003.