Deep Jam: Conversion of Coarse-Grain Parallelism to Instruction-Level and Vector Parallelism for Irregular Applications

Patrick Carribault [1 2 3] Albert Cohen [3] William Jalby [2]

[1] Bull SA, Les Clayes sous Bois, France

[2] LRC ITACA, CEA/DAM and Université de Versailles Saint-Quentin, France

[3] ALCHEMY Group, INRIA Futurs and LRI, Université Paris-Sud, France

Abstract

A number of compute-intensive applications suffer from performance loss due to the lack of instruction-level parallelism in sequences of dependent instructions. This is particularly accurate on wide-issue architectures with large register banks, when the memory hierarchy (locality and bandwidth) is not the dominant bottleneck. We consider two real applications from computational biology and from cryptanalysis, characterized by long sequences of dependent instructions, irregular control-flow and intricate scalar and array dependence patterns. Although these applications exhibit excellent memory locality and branch-prediction behavior, state-of-the-art loop transformations and back-end optimizations are unable to exploit much instruction-level parallelism. We show that good speedups can be achieved through deep jam*, a new transformation of the program control- and data-flow. Deep jam combines scalar and array renaming with a generalized form of recursive unroll-and-jam; it brings together independent instructions across irregular control structures, removing memory-based dependences. 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 charactristics.*

1. Introduction

A wealth of compiler optimizations for high-performance applications address the exploitation of fine-grain parallelism in 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 [17, 6]. Many hardware designs and software solutions address this issue.

Hardware Approaches.

- Multithreading is a flexible solution, either because the program is explicitly threaded, or assuming the compiler can automatically extract parallel loops and asynchronous procedure calls [5, 34], or if the architecture supports speculative thread-level parallelism [33]. In modern processors, simultaneous multithreading [48] is specifically aimed at the filling of idle functional units with independent computations.
- Large and structured instruction windows also enable coarser grain parallelism to be exploited in aggressive superscalar designs [12, 35].
- Load/store speculation and value prediction can also improve out-of-order superscalar execution of dependent instruction sequences [17, 6].
- 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 on reconfigurable logic [27, 51].

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 [1] may improve the effectiveness of back-end scheduling phases: loop fusion and unroll-and-jam combined with scalar promotion [21, 7] is popular in modern compilers. Several authors extended software-pipelining to nested loops, e.g., through hierarchical scheduling steps [25, 49] or modulo-scheduling of outer loops [37]. These techniques apply mostly to regular, static control loop nests. Extension to loops with conditionals may incur severe overheads [29], and none of these approaches handle nested `while` loops. Trace-scheduling [16] and tail-duplication [29, 2] can also increase the amount of fine-grain parallelism in intricate acyclic control-flow, but its ability to convert coarser-grain parallelism is limited.

Independently, *software thread integration* (STI) [14, 45] has been proposed to map multithreaded applications on small embedded devices without preemptive multitasking operating systems.

STI proceeds to 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 [44]. Yet STI does not allow any dependences between the threads being statically interleaved, it only provide rough support for nested conditionals (decision trees) and `while` loops, and neither does it support if-conversion and speculation.

Contributions. This paper presents a new program optimization, called *deep jam*, to convert coarse-grain parallelism into finer-grain instruction or vector parallelism. 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 two real codes, allowing idle functional units to be fed with independent operations, with a low control overhead.

Section 2 introduces the primitive jamming transformations of the control-flow, and variations on these to adapt to dynamic execution profiles, then introduces a first deep jam algorithm. Section 3 recalls scalar and array renaming techniques for irregular loop nests, and describes specific optimizations in the context of deep jam. Section 4 integrates all these analyzes and transformations in a practical deep jam algorithm. Section 5 describes two real applications and their performance inefficiencies, then shows how deep jam can achieve good speedups.

2. Jamming Irregular Control

Figure 1 shows basic control-flow transformations performed by deep jam. In this example, the outer loop can not be fused with other loop to increase ILP in control and data independent instructions. To improve performance, let us unroll the loop. Because it does not carry any dependences, the duplicated body can be rescheduled: step (b) matches pairs of identical control structures in the duplicated body, then step (c) jams `if` conditionals and `while` loops pairwise (respectively). This can be seen as a generalized unroll-and-jam [1] 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.

```
for (i=0; i<10; i++) {
  if (p)
    a[i] = …;
  while (q)
    … = a[i] + …;
}
```

(a) original nest

```
for (i=0; i<10; i+=2) {
  if (p1)
    a[i] = …;
  if (p2)
    a[i+1] = …;
  while (q1)
    … = a[i] + …;
  while (q2)
    … = a[i+1] + …;
}
```

(b) unroll, match

```
for (i=0; i<10; i+=2) {
  if (p1 && p2) {
    a[i] = …;
    a[i+1] = …;
  } else if (p1)
    a[i] = …;
  else if (p2)
    a[i+1] = …;
  while (q1 && q2) {
    … = a[i] + …;
    … = a[i+1] + …;
  }
  while (q1)
    … = a[i] + …;
  while (q2)
    … = a[i+1] + …;
}
```

(c) single jamming stage

Figure 1. Irregular jam example

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. Let us first assume that, *among a pair of threadlets candidate for jamming, no value is produced in one threadlet and consumed in the other one*;[1] it is the case for the example in Figure 1.

Starting from any control statement in a procedure's syntax tree, 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. For this step, STI requires a manual selection by the system designer,[2] whereas we simply select pairs of child nodes with the closest inner control-flow. If few matching pairs can be built this way, and if the parent control node is a loop, unroll it by a factor of two before identifying pairs of threadlets. E.g., in Figure 1.(a), the parent control node `for` has no match, so this step unrolls the `for` loop then selects the `if`-`if` and `while`-`while` pairs, as shown in Figure 1.(b).

2. Rename scalar and array variables to remove all memory-based dependences between threadlets. Assuming there are no flow dependences across threadlets, this is straightforward and eliminates all inter-threadlet dependences. This step is not necessary in Figure 1, due to the absence of dependences carried by the outer loop.

3. Following the STI transformation of the control-flow, fuse each pair of loops and each pair of conditionals [44]. 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 [44]. This corresponds to step (c) in Figure 1.

The simple example in Figure 1 presents a pessimistic case of the fusion of `if` conditionals: in the more general case of a binary conditional with non-empty `then` and `else` branches, four cases result from the fusion in step 3, and each one may benefit from improved fine-grain parallelism. Unfortunately, epilogs resulting from the fusion of unbalanced loops do not benefit from the transformation, in general. Technically, the generalized fusion applied in the last step can be further optimized through the detection of impossible paths, the hoisting of conditionals outside the fused loop [29], and back-end optimizations of decision tree latency [8].

[1] But a pair of threadlets may have memory-based dependences.

[2] This is reasonable in a real-time system design approach [14], but not in our compilation context.

2.2. Jamming External Information and Variants

Jamming decisions depend on dynamic information to choose among several variants, considering the impact of other transformations and architectural features.

Profile and Feedback Data. Deep jam has a chance of bringing actual speedups only when significant parts of the execution trace traverses 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. Nevertheless, Section 5 will show that good speedups require considering feedback from actual executions to tune the application of the previous jamming primitives. 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, except when it would result in simplified control-flow (factoring identical conditions) or when hot basic blocks are control-dominated by cold ones.[3] E.g., `if` conditionals in Figure 1.(b) should probably not be fused.

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. This can be achieved at the cost of one extra branch on longer execution paths (to be amortized when the trip-count increases), as shown in Figure 2.(b).

Besides profile information, feedback from the effectivness 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 [22, 11], we may assume IPC statistics are available for each basic block and for each variant; such statistics can be easily obtained from effective runs and harware counters, or using static estimates [50]. 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 [45].

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

- code duplication is vastly reduced in the predicated implementation of a product-state control automaton;
- deep jam is profitable when coalescing basic blocks from independent threadlets succeeds in filling idle functional units: this effect will be reinforced in a predicated implementation.

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

[3] Then, jamming the cold blocks is necessary to jam the hot ones.

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. Figure 2.(a) gives an example of if-conversion for conditionals and (almost) balanced loops. Notice this is an alternative to `while` jamming in the previous example.

Tail-duplication is often associated with if-conversion to improve software pipelining [1]. 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, like unroll-and-jam is not limited to unroll 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 [39, 46].

```
for (i=0; i<10; i+=2) {
  (p1) a[i] = …;
  (p2) a[i+1] = …;
  while (q1 || q2) {
    (p1) … = a[i] + …;
    (p2) … = a[i+1] + …;
  }
}
```

(a) aggressive if-conversion

```
if (q1) {
  if (q2)
    do {
      … = a[i] + …;
      … = a[i+1] + …;
    } while (q1 && q2)
  do
    … = a[i] + …;
  while (q1)
} else
  while (q2)
    … = a[i+1] + …;
```

(b) jamming short loops

Figure 2. Alternative jamming schemes

2.3. Quantitative Evaluation

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

Practical experiments will target the IA64 architecture. Indeed, the Itanium processor family is an ideal candidate to evaluate deep jam, because of its wide issue/execute/retire rate (6 instructions per cycle) and large register file (128 GP and 128 FP). It provides mechanisms to improve instruction-level parallelism in the presence of irregular control or data flow, like multiple branch predictions per cycle (also exists on the IBM Power5) and predication (similar to a VLIW architecture like Philips TriMedia). It also provides rare features like speculative loads and branch hints.

Jamming Leaf Control Nodes. Each jamming variant may be evaluated with respect to a set of characteristic parameters of the application and architecture. Let us model the fusion of a pair of `while` loops, say `while (p1) {` $S1_1$`; …;` $S1_{i_1}$`; }` and `while (p2) {` $S2_1$`; …;` $S2_{i_2}$`; }`, where i_1 and i_2 denote the number of instructions in each loop body.

We suppose that static and/or feedback-directed analyzes have gathered the following set of parameters: n_1 and n_2 denote the average number of iterations of each loop, IPC_1 and IPC_2 denote the average number of instructions per cycle in the loop bodies (considering all back-end optimizations , as defined in section 2.2). Finally, let W denote the issue width of the processor (6 on the Itanium) and P its branch misprediction penalty.[4] The branch predictor will, on average, mispredict only the last iteration of such loops. An estimate of the number of cycles spent in the unjammed pair of loops is

$$\mathrm{NC}_{1;2} = \frac{n_1 i_1}{\mathrm{IPC}_1} + \frac{n_2 i_2}{\mathrm{IPC}_2} + 2P.$$

The "pessimistic" jamming strategy in Figure 1.(c) bails out from the fused loop as soon as one condition is invalidated. Let $\mathrm{IPC}_{1\&2}$ denote the IPC of the fused part and $n_{1\&2}$ its average number of iterations; assuming the IPC of each epilog is still respectively IPC_1 and IPC_2, a performance estimate for the "pessimistic" strategy is

$$\mathrm{NC}_{1\&2} = \frac{n_{1\&2}(i_1+i_2+1)}{\mathrm{IPC}_{1\&2}} + \frac{(n_1-n_{1\&2})i_1}{\mathrm{IPC}_1} + \frac{(n_2-n_{1\&2})i_2}{\mathrm{IPC}_2} + 3P.$$

The "+1" in the instruction count stands for the computation of the conjunction of the loop conditions. Notice $\mathrm{IPC}_{1\&2}$ may be over-approximated by $\min(\mathrm{IPC}_1+\mathrm{IPC}_2, W)$, which corresponds to an ideal interleaving of instructions from both threadlets.

Considering the "short loop" strategy in Figure 2.(b), 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:

$$\mathrm{NC}_{1\&2_{\mathrm{short}}} = \mathrm{NC}_{1\&2} - 2P.$$

Notice the benefit is only valuable on short loops, since the added control complexity and code size may degrade the applicability of back-end optimizations and instruction cache performance.

Conversely, the "optimistic" jamming strategy in Figure 2.(a) bails out when both conditions are invalidated, predicating the execution of each threadlet. Let $\mathrm{IPC}_{1|2}$ denote the IPC of the fused part and $n_{1|2}$ its average number of iterations; a performance estimate for the "optimistic" strategy is

$$\mathrm{NC}_{1|2} = \frac{n_{1|2}(i_1+i_2+1)}{\mathrm{IPC}_{1|2}} + P.$$

Notice $\mathrm{IPC}_{1|2}$ is probably equal to $\mathrm{IPC}_{1\&2}$.

Eventually, notice the three strategies are almost identical when jamming two instances of the same `for` loop: let $i_1 = i_2 = i$, $n_1 = n_2 = n$, $\mathrm{IPC}_1 = \mathrm{IPC}_2 = \mathrm{IPC}$ and $\mathrm{IPC}_{1\&2} = \mathrm{IPC}_{1|2} = 2\mathrm{IPC}$ (the best case), we have

$$\begin{aligned}
\mathrm{NC}_{1;2} &= 2ni/\mathrm{IPC} + 2P \\
\mathrm{NC}_{1\&2} &= (n(2i+1))/(2\mathrm{IPC}) + 3P \\
\mathrm{NC}_{1\&2_{\mathrm{short}}} &= (n(2i+1))/(2\mathrm{IPC}) + P \\
\mathrm{NC}_{1|2} &= (n(2i+1))/(2\mathrm{IPC}) + P
\end{aligned}$$

Other jamming transformations on leaf control structures follow the same modeling principles.

[4] Such parameters are rough estimates in general, but they happen to be quite effective on the IA64 architecture [8].

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. the instruction counts i_1 and i_2 can be obtained in summing 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.

2.4. 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 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 unrealistic results when jamming large decision trees: special care is needed to reduce branch overhead resulting from the product control-state automaton. We propose to effectively 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.

3. Data Dependence Removal

Software thread integration (STI) targets independent threads only; this is a reasonable simplification for real-time system design [14], but this would kill most jamming opportunities in our compilation context. We thus extend the previous control-flow

[5] As an exception, innermost loops that can be efficiently optimized with traditional software pipelining (and possibly if-conversion) should not be unrolled.

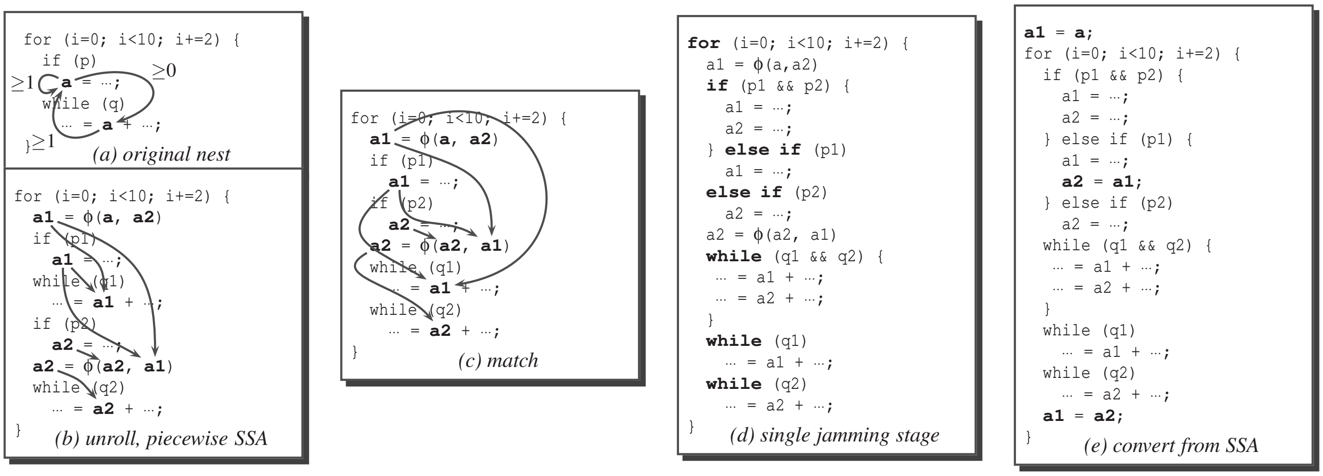

Figure 4. Irregular jam with scalar dependences

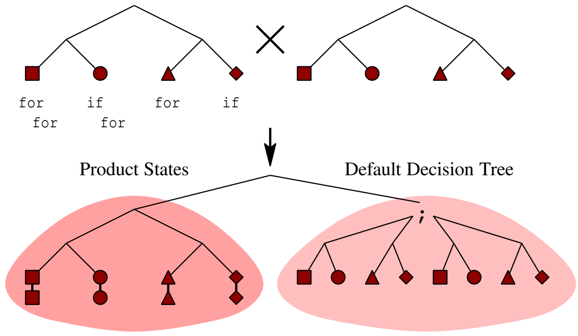

Figure 3. Jamming decision trees

transformations to take dependence information into account, then we plug-in dependence removal techniques to maximize jamming opportunities. This is a major improvement on [44].

3.1. Jamming With Dependences

First of all, it is easy to plug dependence information into the jamming stage described in Section 2.1: step 2 must be restricted to the renaming of scalar and arrays which do not carry any flow of data across threadlets; and step 3 should only be applied to pairs with no inter-threadlet flow dependence. Given these additional constraints, pipeline parallelism may still be available among outer loop iterations: in such a case, deep jam's effect is analogous to the extraction of fine grain parallelism from outer loops through nested software pipelining [37].

This solution requires accurate static analysis to enable a significant amount of coarse-grain to fine-grain parallelism conversion. Also, it is impractical on most programs since the jammed threadlets happen to be clones of the same original conditional or loop (resulting from loop unrolling): most variables incur inter-threadlet dependences, hence hampering fusion. It is thus necessary to resort to more sophisticated scalar and array renaming to safely ignore such memory-based dependences.

3.2. Scalar Renaming

Many dependence removal techniques have been designed in the context of automatic parallelization [15, 26, 47, 24, 9]. 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 static single assignment (SSA). Experience shows that much care must be taken to ensure that the increased parallelism is worth paying an overhead for speculation or dynamic data-flow restoration. Fortunately, in most cases, deep jam reschedules the program in such ways that the overhead of dependence removal techniques can be minimized. Indeed, much of the overhead comes from the interprocessor communications needed for data-flow restoration, which is not an issue for deep jam. In addition, the target of deep jam is still a sequential imperative program, which enables low-cost schemes for data-flow restoration — like DeSSA to convert back from SSA form [13].

Figure 4 revisits the example of the previous section, with a scalar variable `a` producing many intra-loop — unlabeled or 0-labeled edges — and loop-carried dependences — edges with positive distance labels [1]. To reproduce the jamming scheme of Figure 1 despite these dependences, we rename `a` in a *piecewise* variant of the SSA form where only instances of variables in different threadlets get a different name, see Figure 4.(b). Then, steps (c) and (d) can freely reorder and fuse the matching pairs of control structures, without breaking the remaining flow dependences. After all these reordering transformations, the explicit flow of data can be regenerated through DeSSA [13], inserting scalar copies immediately before control-flow merge points associated with ϕ functions, see Figure 4.(e). This SSA-DeSSA transformation has almost no performance overhead since most back-end compilers already operate on SSA form and aggressively optimize away spurious scalar copies [29].

3.3. Array Renaming

Dealing with array dependences is much more complicated. ArraySSA [24] 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.[6] Although DeArraySSA is more complex than DeSSA, the flow of data can be regenerated with low overhead in many cases [24], although precise static analysis of the array data-flow may be required [10].

An example is given in Figure 5. It is similar to the previous example, with an array subscripted with a constant `k` instead of a scalar. Besides renaming array variables and introducing ϕ functions, ArraySSA inserts an "adjunct" array @a_i for each renamed array a_i; these temporary arrays of integers trace the order of assignments to the renamed arrays [24]. Then, the semantics of the ϕ function consists in traversing each array argument, comparing "last write" timestamps stored in the adjunct arrays. This is of course a naive implementation, although static analysis may be required to be able to substitute an optimized form [10, 9]. In this example, the optimization obviously consists in computing only element `k` of the array, since `k` is a invariant.

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 [3, 9], based on array data-flow analysis [15, 10]. These sophisticated expansion schemes have a lower runtime overhead but require accurate static analyzes.

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.

```
int a[n];                           int a1[n], a2[n];
for (i=0; i<10; i+=2) {             (...)
  if (p1)                           if (p1) {
    a[k] = ...;                       a1[k] = ...;
  (...)         array renaming        @a1[k] = i;
  if (p2)          ------>          }
    a[k] = ...;                     if (p2) {
  (...)                               a2[k] = ...;
}                                     @a2[k] = i;
                                    }
                                    a2 = φ(a2, a1);
               naive data-flow restoration ↓
(...)
for (j = 0; j < n; j++)
  a2[j] = (@a2[j] >= @a1[j]) ? a2[j] : a1[j];
             optimized data-flow restoration ↓
(...)
a2[k] = (@a2[k] >= @a1[k]) ? a2[k] : a1[k];
```

Figure 5. Array dependences

3.4. Breaking Dependences Speculatively

Dependences may remain after renaming, 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., [18, 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 exists, and when ad-hoc algorithmic information can be used to avoid squashing the (whole) speculative threadlet.

4. Deep Jam Algorithm

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

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. Nevertheless, 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) [42, 23, 11]. Although we did not yet implement deep jam in a compiler, the previous study allows us to outline the main phases of such a deep jam algorithm.

The deep jam algorithm starts from any control statement in a procedure's syntax tree.

Variant Generation. This step tries iteratively to jam all matching pairs of threadlets, considering all possible variants in a breadth-first fashion. A queue stores the generated trees and their associated current node. Initially, it will contain syntax tree and root node. For each tree in this queue, this step 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 subtrees and add them into the queue. Then for each matching pair, generate a new tree by jamming this pair (a new tree for each jamming variant). This generation takes three steps:

1. Remove scalar output- and anti-dependences across current pair jammed, through piecewise SSA. (Section 3.2.)
2. Remove all array output- and anti-dependences across current matching pair jammed that would not incur high overhead; practically, the amount of dependence removal depends on the accuracy of static analyzes and on the sophistication of the expansion techniques employed. (Section 3.3.)
3. Apply DeSSA: convert from SSA form.

Each time a tree is enqueued, it is also inserted into a set which gathers all candidate codes.

Profitability evaluation. For each tree in the set of candidate codes, execute it or estimate its IPC. (Section 2.2.) With these measurements, run an inner-to-outer profitability analysis. (Section 2.3.)

Selection. Choose the tree with the highest profitability.

[6] Unlike array expansion [15, 9] and privatization [47].

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

Overall, 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. Overall, our experiments will show that it can bring strong speedups, but may also suffer from dynamic performance variations.

5. Experiments

Let us study two real compute-intensive applications characterized by long sequences of dependent instructions, irregular control-flow and intricate scalar and array dependence patterns. 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 [32], which lead to a full collision in August 2004 [20, 4]. 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 a addition with the initial tuple, the 160-bits 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 Itanium2 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. The generated assembly code appears to be cluttered with `nops` and stop bits (IA64 fetch cycle delimiters). This hints towards the lack of ILP in chains of dependent instructions. This analysis is confirmed by hardware counters which detect a large rate of pipeline stalls due to register-to-register dependences:

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

Bitwise operations in each compression round are indeed fairly sequential, and because of the complex control flow, the compiler could not reschedule the loops to expose more ILP. Scalar promotion and vectorization have only skimmed the dependence graph around its critical path, hardening the scheduler's task of filling functional units with useful operations.

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, 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 piecewise SSA. One array of 80 elements needs to be renamed to remove output and anti-dependences; the corresponding data-flow restoration scheme is analogous to the example in Figure 5 since most accesses are either constant or loop invariant; if-conversion is applied to the resulting conditional array copies. 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 from the control-flow graph structure. For example, the inner loop is jammed with its matching pair using the "pessimistic" strategy in Figure 1.(c), 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 [19], 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). In practical searches, k can be as large as $m/2$. Assuming an online search, the pattern is known and can be pre-processed to speedup the search, but the text may not. AMBDN/BPM is a key contribution to the pattern matching domain, since it combines dynamic programming, filtering and bit-parallelism [31]. 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 [30, 31].

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 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 notice that the dynamic computation of the skip between two consecutive windows yields several control and data dependences. Any jamming scheme needs to speculatively break those dependences. Thanks to domain-specific knowledge, we know that underapproximating this distance is a conservative solution (yielding lower performances but still covering all possible matches). Figure 6 sketches a speculative jamming scheme where the position of the second window is estimated at each iteration of the outer (unrolled) loop.

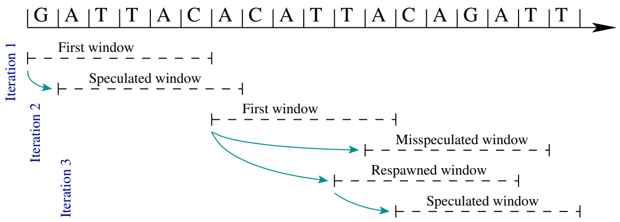

Figure 6. Jamming windows speculatively

Empirical experiments show that, in general, the distribution of skip distances matches the histogram in Figure 7. The associated speedup curve corresponds to a typical deep jam speedup obtained with a fixed speculative skip, ranging from 1 to the maximal distance. From these encouraging results, we designed a skip prediction mechanism, maximizing the skip distance while keeping the level of misspeculations under a threshold. The speculated skip is updated every iteration of the outer loop with respect to the success of the previous speculation: it is incremented if the distance was underestimated multiple (consecutive) times, set to the effective distance if it was overestimated (the backward traversal of the second window was squashed in that case), and left unchanged otherwise.

Figure 7. Skip distances vs. speedup

Then, a large number of scalar dependences are eliminated by piecewise SSA, but no array dependences need to be removed. No dependences remain across threadlets (except those speculatively broken), allowing a one-to-one fusion of every matching pair of conditionals and inner loops.

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 2.(a) 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 2.(b). 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 2.2.

Figure 8 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.

6. Conclusion

This paper 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 and up to 58.9% on a computational biology application. 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.

In the short term, deep jam is very appealing for domain-specific program generators [28]; we believe expert knowledge (from the programmer) will dramatically smooth the implementation challenges, compared with a general-purpose compiler optimization framework. Beyond implementation and evaluation in a compiler, it would be interesting to evaluate deep jam for grid processors [39, 46], reconfigurable computing and hardware synthesis [43], and custom extensions to VLIW processors [40, 41]. 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 [44]. Since deep jam revisits several automatic parallelization techniques for irregular programs, coupling it with hybrid static-dynamic analyzes [38] may improve its effectiveness.

Acknowledgments. We would like to thank Antoine Joux for providing the SHA-0 cryptanalysis code, Christophe Lemuet and Jean-Thomas Acquaviva for their contributions to the manual optimization of this code, and Gonzalo Navarro for providing the string-matching application.

References

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

[2] 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.

[3] D. Barthou, A. Cohen, and J.-F. Collard. Maximal static expansion. In *25^{th} ACM Symp. on Principles of Programming Languages (PoPL'98)*, pages 98–106, San Diego, California, Jan. 1998.

[4] 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.

[5] W. Blume, R. Eigenmann, K. Faigin, J. Grout, J. Hoeflinger, D. Padua, P. Petersen, W. Pottenger, L. Rauchwerger, P. Tu, and S. Weatherford. Parallel programming with Polaris. *IEEE Computer*, 29(12):78–82, Dec. 1996.

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

[7] 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.

[8] P. Carribault, C. Lemuet, J.-T. Acquaviva, A. Cohen, and W. Jalby. Branch strategies to optimize decision trees for wide-issue architectures. In *Workshop on Languages and Compilers for Parallel Computing (LCPC'04)*, LNCS, West Lafayette, Indiana, Sept. 2004.

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

[10] J.-F. Collard. The advantages of reaching definition analyses in Array (S)SA. In *11^{th} Workshop on Languages and Compilers for Parallel Computing*, number 1656 in LNCS, pages 338–352, Chapel Hill, North Carolina, Aug. 1998. Springer-Verlag.

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

[12] A. Cristal, O. J. Santana, M. Valero, and J. F. Martínez. Toward kilo-instruction processors. *ACM Trans. on Architecture and Code Optimization*, 1(4):389–417, 2004.

[13] 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*, 13(4):451–490, Oct. 1991.

[14] 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.

[15] P. Feautrier. Array expansion. In *ACM Int. Conf. on Supercomputing*, pages 429–441, St. Malo, France, July 1988.

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

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

[18] M. Griebl and J.-F. Collard. Generation of synchronous code for automatic parallelization of `while` loops. In S. Haridi, K. Ali, and P. Magnusson, editors, *Euro-Par'95*, volume 966 of *LNCS*, pages 315–326. Springer-Verlag, 1995.

[19] H. Hyyrö and G. Navarro. Faster bit-parallel approximate string matching. In *Proc. Symp. on Combinatorial Pattern Matching (CPM'02)*, number 2373 in LNCS, pages 203–22. Springer-Verlag, 2002.

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

[21] K. Kennedy and K. McKinley. Maximizing loop parallelism and improving data locality via loop fusion and distribution. In *Languages and Compilers for Parallel Computing*, pages 301–320, Portland, 1993.

[22] 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.

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

[24] K. Knobe and V. Sarkar. Array SSA form and its use in parallelization. In *25^{th} ACM Symp. on Principles of Programming Languages*, pages 107–120, San Diego, CA, Jan. 1998.

English dictionary.

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%

LaTeX document.

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%

DNA of Buchnera bacterium.

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%

DNA of bacillus anthracis str. Ames.

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%

Figure 8. Performances of three deep jam variants on ABNDM/BPM

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

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

[27] 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.

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

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

[30] G. Myers. A fast bit-vector algorithm for approximate string matching based on dynamic programming. *J. of the ACM*, 46(3):395–415, 1999.

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

[32] N. I. of Standards and Technologies, editors. *Secure Hash Standard*, volume FIPS-180. Information Processing Standards Publication, may 1993.

[33] J. Oplinger, D. Heine, and M. S. Lam. In search of speculative thread-level parallelism. In *Parallel Architectures and Compilation Techniques (PACT'99)*, pages 303–313, Newport Beach, CA, Oct. 1999.

[34] G.-R. Perrin and A. Darte, editors. *The Data Parallel Programming Model: Foundations, HPF Realization, and Scientific Applications*. Number 1132 in LNCS. Springer-Verlag, 1996.

[35] 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.

[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*, 10(2):160–180, 1999.

[37] 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)*, pages 163–174, Mar. 2004.

[38] S. Rus, D. Zhang, and L. Rauchwerger. The value evolution graph and its use in memory reference analysis. In *Parallel Architectures and Compilation Techniques (PACT'04)*, Juan-les-Pins, France, Oct. 2004.

[39] 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.

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

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

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

[43] 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.

[44] 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.

[45] 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.

[46] 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.

[47] P. Tu and D. Padua. Automatic array privatization. In *6thWorkshop on Languages and Compilers for Parallel Computing*, number 768 in LNCS, pages 500–521, Portland, Oregon, Aug. 1993.

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

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

[50] 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.

[51] 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.