VISTA: The Visual Interface for Scheduling Transformations and Analysis*

TECHNICAL REPORT 94-22

Steven Novack and Alexandru Nicolau
Department of Information and Computer Science
University of California
Irvine, CA 92717

*Appears in the proceedings of Sixth Annual Workshop on Languages and Compilers for Parallelism. This work was supported in part by NSF grant CCR8704367 and ONR grant N0001486K0215.

VISTA: The Visual Interface for Scheduling Transformations and Analysis*

Steven Novack and Alexandru Nicolau
Department of Information and Computer Science
University of California
Irvine, CA 92717

Abstract

VISTA is a visually oriented, interactive environment for parallelizing sequential programs at the instruction level for execution on fine-grain architectures. Fully automatic parallelization techniques often perform well, but may not be able to achieve the strict performance and code size requirements needed for some critical applications. In such cases, manual manipulation by an expert user can often provide enough improvements in the parallelization process to meet the requirements of the application. Using VISTA, an expert user fine-tunes the parallelization process by providing rules and directives to the system in response to graphical and numeric feedback provided by the system.

1 Introduction

The Visual Interface for Scheduling Transformations and Analysis (VISTA) is a visually oriented, interactive environment for the semi-automatic parallelization of sequential programs at the instruction level for execution on fine-grain architectures. Fully automatic parallelizing compilers (e.g. [8, 7, 10, 5]) perform well on average and produce generally good results, but in some cases may fail to extract a sufficient level of parallelism from a given program to achieve the high performance required for the application while satisfying the physical constraints (e.g. allowable memory usage) of the target machine. For many applications, such as embedded systems in aircraft, meeting specific cost and performance constraints may be strictly necessary for the application to work at all. For such applications, even small improvements over fully automatic techniques obtained by manual manipulation of the parallelization process can be the enabling factor for the feasibility of a given project.

The failure of a fully automatic compiler to achieve a desired level of performance, even though such a level is in fact achievable, is usually a consequence of two problems inherent to automatic compilation: generally good heuristics for solving NP-hard problems can fail to perform adequately for some specific cases, and a lack of application specific knowledge can force the compiler to make overly conservative assumptions. The NP-hard characteristic of resource-constrained scheduling and scheduling in the presence of conditional branches suggests that there is no "best" general solution for parallelizing programs for execution on actual machines. Even though the heuristics employed by fully automated compilers for dealing with these problems may perform well on average, for any particular application, an interactive tool like VISTA will often enable an expert user to produce schedules that are significantly better than those produced by the available fully automated compilers. Furthermore, a lack of application specific knowledge can greatly degrade the performance of programs by forcing

*Appears in the proceedings of Sixth Annual Workshop on Languages and Compilers for Parallelism. This work was supported in part by NSF grant CCR8704367 and ONR grant N0001486K0215.

the compiler to make overly conservative decisions in such areas as the disambiguation of indirect memory references needed for instruction scheduling and the estimation of path execution probabilities necessary for making good speculative scheduling choices. Clearly, manual intervention by the user to provide application specific knowledge or scheduling direction need not (and should not) be applied at every decision point during the parallelization process. The fact that even large programs spend most of their time executing in a small fraction of their code (i.e. in loops or recursive routines) suggests that such special manual attention need only be applied at relatively few stages of compilation (e.g. at loops on critical paths) and then only selectively when the compiler is not able to make an obviously superior scheduling choice or would otherwise have to make an overly conservative decision that might adversely affect the performance of the program.

The VISTA environment allows the user to fine-tune the parallelization process using problem specific knowledge, human intuition, and trial-and-error approaches, when and where needed, by providing decisions, directives and rules to the system in response to system queries or at user specified decision points. This fine-tuning can take two forms. First, the user can provide information to the parallelization system that can not always be computed efficiently (or at all), such as identification of critical paths and bottlenecks, disambiguation of pointer and indirect references, and control path execution probabilities. Second, during the parallelization process, there may be decision points at which the system is not able to determine a clearly superior course of action. Some examples are cost vs. performance trade-offs (e.g. code size and compile time vs. parallelism), high-level transformation choices (e.g. which level to pipeline[1] in a nested loop), and choosing alternative scheduling strategies (e.g. transformation ordering heuristics and speculative scheduling strategies[2]). These decisions may be different at different points in the program and at different stages of the parallelization process.

Achieving the abovementioned functionality requires three things: an ability to visualize the parallelization process, a system of parallelizing program transformations that is powerful enough to provide maximum parallelism relative to system specific constraints while remaining efficient and flexible enough to allow for arbitrary scheduling approaches and cost vs. performance trade-offs in an interactive environment, and finally, mechanisms for analyzing and anticipating the effects of such transformations during parallelization to provide the user with useful information and feedback upon which to base scheduling decisions. For small programs, Control Flow Graphs (CFG's) can be graphically displayed to provide a natural, visual representation of programs upon which a system of parallelizing transformations, such as Percolation Scheduling (PS)[9], can be performed. PS parallelizes programs by repeated application of a pair of transformations, called move-op and move-cj, that move an operation or conditional jump up one instruction[3] in the CFG while preserving the semantics of control and data flow. These transformations need not be applied in any specific order or exhaustively, thus, PS provides the flexibility to implement arbitrary scheduling heuristics.[4] Performing PS on CFG's can provably extract all of the parallelism available in a program[2]). However, as program size increases, CFG's become too unwieldy and complicated, especially after parallelization, to be useful for visualization and inefficiencies inherent to straight PS can begin to have significant detrimental effects on compilation time and code size. These inefficiencies derive mainly from the strictly incremental application of transformations and significant code explosion caused by loop unrolling and parallelizing code containing

[1] *Loop pipelining* refers to overlapping the execution of successive iterations of a loop.

[2] Determining when to move operations above conditionals.

[3] A VLIW instruction is a set of risc-like operations that can be executed in parallel, possibly containing multiple conditional jumps that combine to yield a single control path.

[4] The transformations themselves ensure the preservation of correct semantics.

multiple control paths.

To overcome these problems, VISTA incorporates a hierarchical representation of the CFG based on Hierarchical Task Graphs (HTG's)[6] and an enhancement of PS called Trailblazing PS (TiPS)[11] that overcomes the inefficiencies of PS while retaining its power and flexibility. HTG's partition the CFG into a hierarchy of subgraphs, usually with a direct correspondence to the hierarchy of the original high-level representation of the program, thus providing a visualizable and understandable structure to the instruction level representation being parallelized. TiPS also exploits this structure by extending the PS core transformations to navigate through the HTG hierarchy. At the lowest level sub-graphs in the hierarchy, TiPS is able to perform the same fine-grained transformations as normal PS, while at higher levels, TiPS is able to move operations across large blocks of code in constant time, including loops, past which normal PS is unable to move operations at all. This non-incremental code motion itself improves efficiency over PS by bypassing multiple instructions in constant time, but also allows code explosion to be controlled by splitting control paths only when necessary to extract more parallelism,[5] and then only if the cost of doing so is justified by the performance thus achieved. TiPS attempts to move operations at the "highest" possible level, where they can bypass multiple nodes at lower levels in constant time, while moving operations to, or keeping them at, lower levels when needed to expose more parallelism. In this fashion, TiPS is able to provide efficient code motion without sacrificing the "completeness" of normal PS and even enables some code motion not directly possible in normal PS.

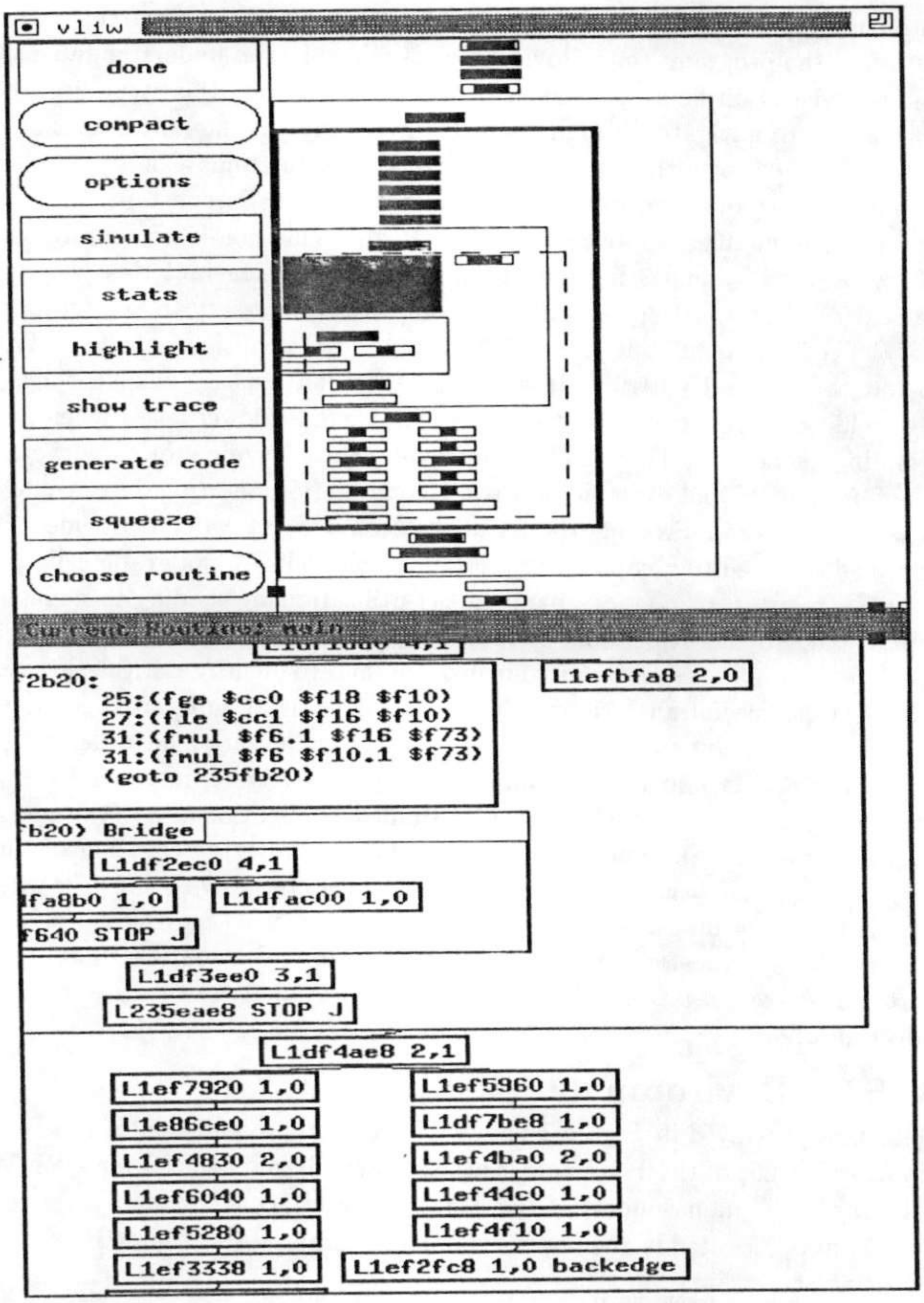
In order to assist the user in directing the parallelization process, VISTA provides visually-oriented analysis and (parallel) code editing capabilities. For example, VISTA can graphically portray resource (e.g. functional unit, register, or memory bandwidth) utilization by shading in nodes in the HTG by an amount proportional to the utilization at that point. This feature helps the user to monitor the effects of different transformations on resource utilization and to identify critical paths and bottlenecks within the code. Another useful analysis capability of VISTA is an integrated simulator that can be used to report dynamic speedups at successive stages of parallelization in order to gauge the effect of the transformations. This information can also be graphically represented by shading in nodes in the HTG by an amount proportional to the normalized execution time (the percentage of total time spent executing each node), thus providing a global, visual representation of the critical paths throughout the program. The visual editing capabilities of VISTA provide the user with a "point-and-click" environment for performing transformations at any level, from moving individual operations to initiating a loop pipelining transformation or parallelizing an entire routine. The remaining sections of this paper provide an overview of the VISTA interface (Section 2) and its use in fine-tuning specific applications (Section 3).

2 The VISTA Environment

This VISTA interface portrayed in Figure 1 shows the HTG representation of the Discrete Ordinates Transport loop (number 20) of the Livermore Kernels, partially parallelized for a VLIW machine with four functional units. The main window of the graphical interface consists of three regions. The first region, called the *canvas*[6], located in the upper right-hand corner of the display shows the entire routine

[5]When moving conditionals or when moving operations that can move more on one control path than on another, control paths may need to be duplicated, or *split*, in order to preserve the semantics of control flow. Normal PS will always perform such splitting when moving operations along multiple control paths, even if it is not semantically necessary (e.g. when moving an operation across an entire if-then-else block).

[6]The reader familiar with the X environment will recognize many of these names (and regions) as being widgets from the X Athena Widget set.



```
25:(fge $cc0 $f18 $f10)
27:(fle $cc1 $f16 $f10)
31:(fmul $f6.1 $f16 $f73)
31:(fmul $f6 $f10.1 $f73)
(goto 235fb20)
```

Figure 1: HTG after automatic compaction without loop pipelining. Functional unit utilization is highlighted in the canvas.

represented as a tree[7] of nodes. The dashed-line rectangle surrounding a portion of the canvas is called the *slider*. The second region, the *porthole*, located at the bottom of the display shows a magnified view of the rectangular portion of the canvas enclosed by the slider.

Each rectangle shown in the porthole represents a node in the HTG and is labelled by a unique number (actually, the memory address of the node itself). HTG's were originally presented for use at the coarse-grain level but we have adapted them for use at the instruction-level. An HTG is a directed acyclic graph containing five types of node: START and STOP nodes indicating the entry and exit of HTG's respectively, Simple nodes that, for our purposes, represent VLIW instructions, Compound nodes representing sub-HTG's which we use to represent if-then-else blocks, and finally, Loop nodes that represent loops whose bodies are sub-HTG's. Rectangles other than the slider in the canvas or porthole that contain other rectangles represent either compound or loop nodes, and rectangles that contain no others represent either simple nodes or STOP nodes. Loop nodes containing inner loops are distinguished from the other nodes in the canvas by a rectangle drawn with thick lines as in Figure 1.

When moving an operation, op, TiPS is able to determine whether or not a dependency exists between op and a compound or loop node, say B, without visiting any node within HTG(B) (the sub-HTG of B).[8] If there is no dependency or if the dependency can be removed (e.g. by renaming[3]), then op can move non-incrementally across B without visiting any nodes within HTG(B). For compound nodes containing (possibly nested) if-then-else blocks, this can provide significant savings over PS in terms of compile time and code size since the worst case cost of both for moving the same operation across the if-then-else block using PS is exponential in the number conditionals within the block. The motion across loops (represented as loop nodes) provided by TiPS is not even possible using PS. Since compound nodes and loop nodes can both act as "bridges" across their sub-HTG's, we refer to them collectively as *bridge nodes* or just *bridges*. Given a bridge B, HTG(B) is referred to as the *region bridged by B* or just *bridged region* if B is understood. A variety of information about each node can be shown in the corresponding rectangle in the porthole of the VISTA display, including the entire operation tree of simple nodes or the entire region bridged by bridge nodes. In Figure 1, all bridged regions are displayed this way and the operations of one simple node are shown as an example, but in general, we just characterize simple nodes by the number of operations contained within them and any special characteristics that distinguish them from the others.[9] At the right-hand side of each rectangle is a pair of numbers, "X,Y", where X is the total number of operations in the node and Y is the number of operations in the node that are conditional jumps. Some nodes are distinguished with further labels, such as "loop head" if the node is the head of a loop or "backedge" if at least one of the successors of the node is a backedge, "Bridge" if the node is a bridge and "Stop" if it is a STOP node (START nodes are not explicitly shown).

The final region of the main display, called the *control region*, is shown in the upper left-hand corner of the display. The buttons in this region are used to provide global control of the parallelization process when the compiler is in interactive mode. Additional, finer-grain control is provided via commands and menus associated with the nodes themselves. The functionality provided by these controls fall generally into two categories of transformations: parallelism exposing and parallelizing. Parallelism exposing

[7]For the sake of simplicity, we currently use a tree representation for the program, therefore, the multiple predecessors of join-points and backedges are not explicitly shown.

[8]This is accomplished by associating *definition* and *use* sets with B that indicate which variables are defined or used within HTG(B). If an operation shares no definitions or uses with B, then it has no dependency on any operation in HTG(B).

[9]For the purposes of this paper, the operation-level details of the code being parallelized are unimportant.

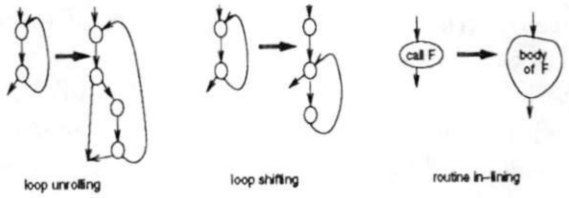


Figure 2: Exposing Parallelism

transformations (see Figure 2) consist of loop unrolling (inserting one or more successive iterations of a loop into its body),[10] loop shifting ("unwinding" a loop so that its head becomes a true successor of each of its predecessors), and routine in-lining (replacing a CALL by the body of the routine being called). These transformations do not in general improve performance (although, each in-lining does eliminate a CALL and a RETURN); however, they do expose new operations that can be scheduled in parallel with existing operations, thereby allowing for improved performance at the cost of increased code size and compile time. Parallelization is affected by a hierarchy of transformations. At the lowest level are *trailblaze* (the TiPS equivalent of the PS move-op transformation) and *move-cj* (same for both PS and TiPS) which move individual operations or conditional jumps, respectively. Next higher up is the *schedule* transformation that moves operations in a user-specified order, using *trailblaze* and *move-cj*, toward the node currently being scheduled, say n, until no further operations can move into n. Operations can be prevented from moving due to data dependencies, resource constraints, and artificial limitations such as cost vs. performance trade-offs. The highest level parallelizing transformation is the *compact* transformation which traverses a specified sub-graph of the program in a top-down fashion while applying the *schedule* transformation to each node visited (the specified sub-graph may be the entire program HTG).

3 Fine-tuning parallelization

These transformations are usually performed automatically by the system; however, control is available for manually applying them as needed, at any level, in order to fine-tune specific applications. The VISTA interface can be used to provide interactive disambiguation and speculative scheduling decisions; however, in this section, we will focus on the ability of VISTA to tune cost vs. performance trade-offs. Tuning cost vs. performance trade-offs is one of the more intractable problems for fully automated parallelizing compilers since acceptable ratios of cost to performance depend entirely on the application and architecture at hand, and achieving this ratio is itself an NP-hard problem. For example, without any restrictions on program size, it is usually a good idea to fully loop-pipeline the inner loops of programs. Loop Pipelining consists of repeatedly exposing parallelism within an inner loop (using unrolling as in [10] or shifting as in [4]) and compacting the resultant loop until no more parallelism can be extracted from the loop. Loop Pipelining usually has the effect of significantly improving performance at the expense of increased program size. When there are limits on program size, VISTA allows loop pipelining to be terminated when the user determines that either enough parallelism has already been exploited or the point of diminishing returns of the cost vs. performance ratio has been reached. To facilitate the making of decisions, such as these, based on performance goals and cost vs.

[10] In our compiler, if unrolling is applied more than once, iterations from the original (rather than the current, previously unrolled) loop are inserted.

performance trade-offs, VISTA has mechanisms for quantitatively and graphically providing feedback to the user about the effects of transformations that have been applied to the program. Quantitative feedback can take the form of reports on program size, static speedup computed as the average degree of parallelism over a specified region of code, or dynamic speedup, the ratio of sequential to parallel execution times determined by simulating the code before and after applying any transformations. For instance, simulation (initiated by the SIMULATION button) of the schedule in Figure 1 yields a speedup of 2.41 (reported using the STATISTICS button).

While quantitative feedback is primarily useful for analyzing the effect of previously applied transformations, graphical feedback in the form of visual representations of resource utilization and normalized execution times help the user to anticipate the effect of future transformations. Graphical representation, referred to as *highlighting*, is provided by shading the rectangles in the canvas by an amount proportional to the specified resource utilization or normalized execution time at that point in the program. For example, the canvas in Figure 1 shows functional unit utilization (i.e. the percentage of functional units that are used by the operations in each node). By highlighting resource utilization, the user can determine where parallelism exposing transformations might be useful — if no resources are available, then, in general, there is no point in exposing new operations for movement.[11] For instance the availability of unutilized resources, indicated by the unshaded areas of the inner loop nodes displayed in the canvas of Figure 1, suggests that applying loop shifting and/or unrolling on the inner loop would probably improve performance. The amount of performance gained and its cost in increased code size depends on which technique is used, the architecture, and the characteristics of the loop itself. In general, when combined with a loop restructuring technique such as Perfect Pipelining[1, 10], loop unrolling can achieve maximum performance by allowing iterations to be scheduled successively until resources are utilized as well as possible,[12] but can potentially result in a significant amount of code explosion. On the other hand, loop shifting has the advantage of keeping the number of iterations constant (with somewhat less code explosion), but at the cost of arbitrarily limiting the number of operations within the loop, and therefore the maximum possible resource utilization since the fixed number of operations may not be evenly divided among the available functional units. For example, in Figure 3, the user might decide to shift rather than unroll the loop due to the lesser code explosion penalty and the likelihood that unrolling would not perform significantly better than shifting for this particular loop running on the given architecture. The intuition behind this decision is that since the architecture is relatively "narrow" with respect to the number of operations in the loop (4 functional units compared to 52 operations), then the penalty for shifting, of not being able to evenly divide the operations by the number of functional units, is likely to be small. Figure 3 shows the inner loop shifted 7 times which resulted in a 17% increase in speedup (to 2.82) at the cost of a 15% increase in code size relative to the schedule in Figure 1. While the decision to use shifting instead of unrolling may be appropriate for this particular loop running on a "four-wide" VLIW architecture, results in [10] show that for different loops and/or different architectures the performance penalty of shifting can be significant, in which case unrolling might be a better choice. Whether unrolling, shifting, or some combination of the two is best for a given application depends entirely on the characteristics of the application itself and the target architecture. Any automatic, deterministic heuristic for applying these transformations would, by necessity, be sub-optimal for some application/architecture combinations for

[11]Although, in some cases, opportunities for redundant operation removal (e.g. load after store elimination or constant folding) might be exposed and therefore may make the parallelism exposing transformation worthwhile even in the absence of available resources.

[12]Data dependencies that cross iteration boundaries can prevent full utilization.

which an expert user would nevertheless be able to find a superior solution if provided with sufficiently powerful tools for understanding and directing the parallelization process.

When multiple opportunities for parallelism exposing transformations have been identified, it may be necessary to choose from amongst them if there are limits on code size. An example of this is shown in Figure 3 wherein loop shifting and/or unrolling can be performed on either of the loop heads.[13] At this stage, it is useful to highlight the normalized execution time of the nodes in the program in order to provide a global picture of the critical paths through the code. Given this information, the user can choose which of the potential parallelism exposing and parallelizing transformations to apply based on his estimation of their likely affect on the critical paths and the relative cost vs. performance benefits of each. Then, using quantitative feedback, the user can analyze the actual costs and performance thus obtained. For example, by highlighting the normalized execution time in the canvas of Figure 3 we can see that the loop on the left is more critical than the one on the right, and therefore the next parallelism exposing transformation (and compaction) should occur on the left-most loop head. By shifting and compacting another 4 times along the left-most loop a speedup of 3.2 would be achieved, providing an incremental improvement of 14% over the schedule in Figure 3 at the incremental cost of a further 3% increase in code size. After doing this, highlighting functional unit utilization would show that resources are almost completely utilized. If it turns out that the cost vs. performance ratio is not acceptable (a determination which is entirely application dependent), then the transformations can be "undone" by replacing the outer-most HTG that was affected by the transformations with a copy saved prior to performing the transformations (VISTA allows the user to save any HTG in its current state for possible restoration at a later time, but only allows restoration when semantics would be preserved by doing so).

For the example shown in Figures 1 and 3, fully automatic techniques, such as those presented in [4, 10], would usually result in much greater code explosion than obtained using VISTA, but without any significant improvements in speedup since, in the final schedule described above, resource utilization obtained with VISTA approaches 100%. In fact, depending on the choice of scheduling heuristics used by either of these (or any other) automatic techniques, speedup might actually degrade because of the added pressure on functional units caused by too much code explosion (i.e. more operations compete for the same resources). For example, the same scheduling heuristics used to create the nearly optimal results presented in [10] for some loops, would, for this particular loop, have resulted in a speedup of 2.9 at a cost in code size of over 1000 instructions (as compared to the speedup of 3.2 at a cost of 42 instructions produced using VISTA).

By selectively applying exposing and parallelizing transformations based on application-specific knowledge, intuition, and feedback from the VISTA system, the user can often fine-tune the parallelization process to provide cost vs. performance trade-offs superior to those produced by the available fully automatic systems. The problems with fully automatic techniques can be alleviated somewhat with better scheduling heuristics; however, due to the complicated interactions between the different heuristics used for dealing with the various NP-hard problems involved in compiling for fine-grain parallelism, human intervention can often be the deciding factor for achieving the necessary cost and performance goals.

[13]Notice that the loop has become irreducible as a result of shifting a node that contained a conditional branch.

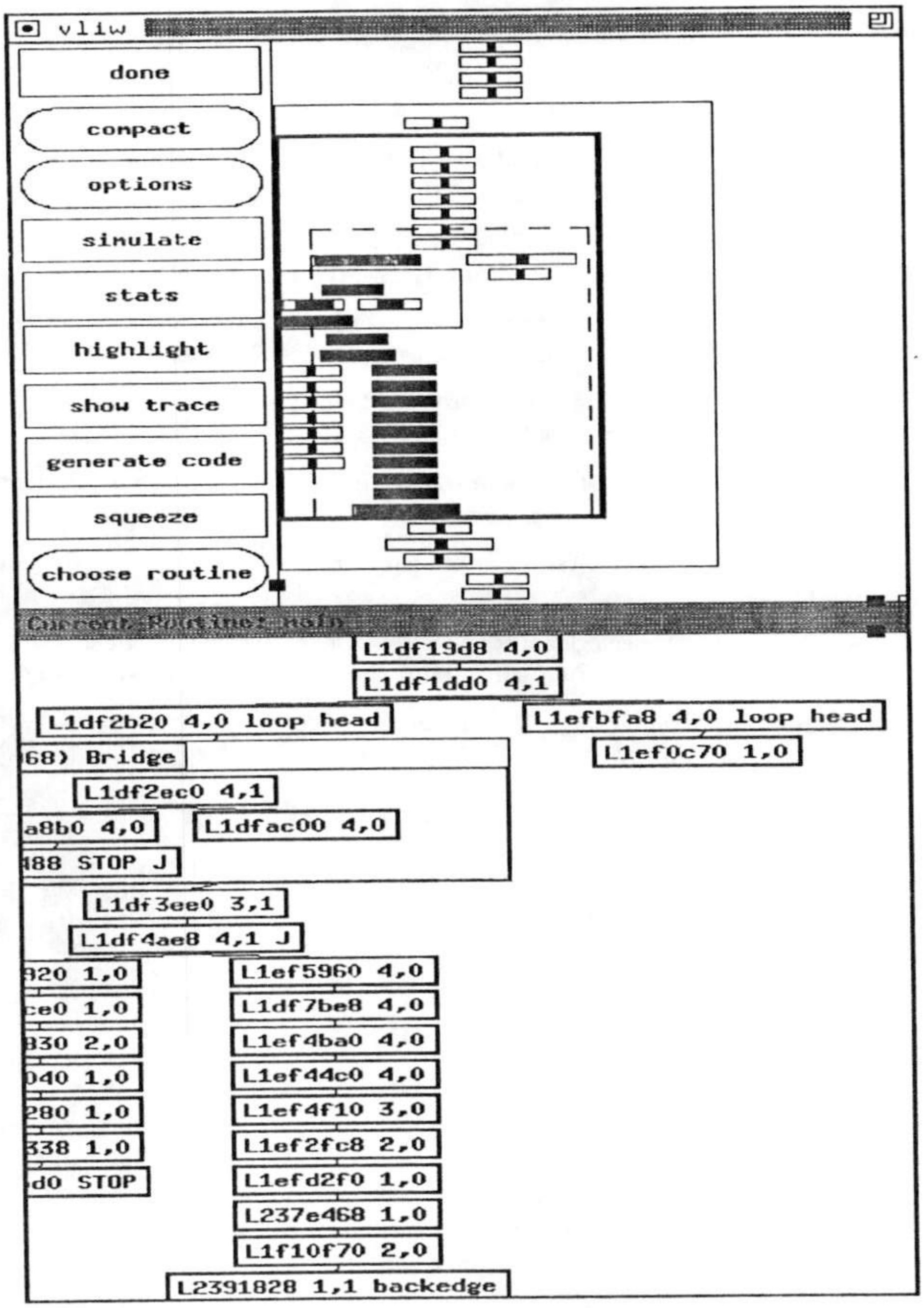


Figure 3: HTG after user directed shifting and compaction. Normalized execution time is highlighted in the canvas.

References

[1] A. Aiken and A. Nicolau. Perfect pipelining: A new loop parallelization technique. In *Proceedings of the 1988 European Symposium on Programming*. Springer Verlag Lecture Notes in Computer Science no. 300, March 1988.

[2] A. S. Aiken. *Compaction-Based Parallelization*. PhD thesis, Cornell University, 1988.

[3] R. Cytron and J. Ferrante. What's in a name? In *Proc. of the 1987 Int'l Conf. on Parallel Processing*, pages 19–27, August 1987.

[4] K. Ebcioglu and T. Nakatani. A new compilation technique for parallelizing loops with unpredictable branches on a vliw architecture. In *Proceedings of the 2nd Workshop on Programming Languages and Compilers for Parallel Computing*, Urbana, IL, 1989.

[5] J. R. Ellis. *Bulldog—A Compiler for VLIW Architectures*. MIT Press, 1986.

[6] M. Girkar and C. Polychronopoulos. Automatic extraction of functional parallelism from ordinary programs. *IEEE Transactions on Parallel and Distributed Systems*, 3(2):166–178, March 1992.

[7] S. Moon and K. Ebcioglu. An efficient resource constrained global scheduling technique for superscalar and vliw processors. Technical report, IBM, 1992.

[8] T. Nakatani and K. Ebcioglu. Using a lookahead window in a compaction-based parallelizing compiler. In *Proceedings of the 23rd Annual International Symposium on Microarchitecture*, 1990.

[9] A. Nicolau. Uniform parallelism exploitation in ordinary programs. In *Proc. of the 1985 Int'l Conf. on Parallel Processing*, 1985.

[10] S. Novack and A. Nicolau. An efficient global resource constrained technique for exploiting instruction level parallelism. In *Proc. of the 1992 Int'l Conf. on Parallel Processing*, St. Charles, IL, August 1992.

[11] S. Novack and A. Nicolau. Trailblazing: A hierarchical approach to percolation scheduling. Technical Report TR-92-56, Univ. of Calif. at Irvine, 1992. Also appears in the *Proc. of the 1993 Int'l Conf. on Parallel Processing*, St. Charles, IL, August 1993.