Spacewalker: Automated Design Space Exploration for Embedded Computer Systems

Greg Snider
HP Laboratories Palo Alto
HPL-2001-220
September 10th , 2001*

design space exploration, VLIW, systolic array, cache

This paper addresses the problem of automated design of a computer system for an embedded application. The computer system to be designed consists of a VLIW processor and/or a customized systolic array, along with a cache subsystem comprising a data cache, instruction cache and second-level unified cache. Several algorithms for "walking" the design space are described, and experimental results of custom designed systems for two applications are presented.

* Internal Accession Date Only Approved for External Publication

1 Problem

This paper addresses the problem of automated design of a computer system for an embedded application. The computer system to be designed consists of a VLIW processor and/or a customized systolic array, along with a cache subsystem comprising a data cache, instruction cache and a second-level unified cache (figure 1). The VLIW processor connects via a single port to the instruction cache and through one or more ports to the data cache. The systolic array reads and writes data only through ports to the unified cache. If the VLIW and systolic array are both present, the systolic array is configured as a slave to the VLIW and is controlled through a local memory; the VLIW invokes the systolic array through a procedure-call-like mechanism, and hence the VLIW and systolic array never execute instructions concurrently.

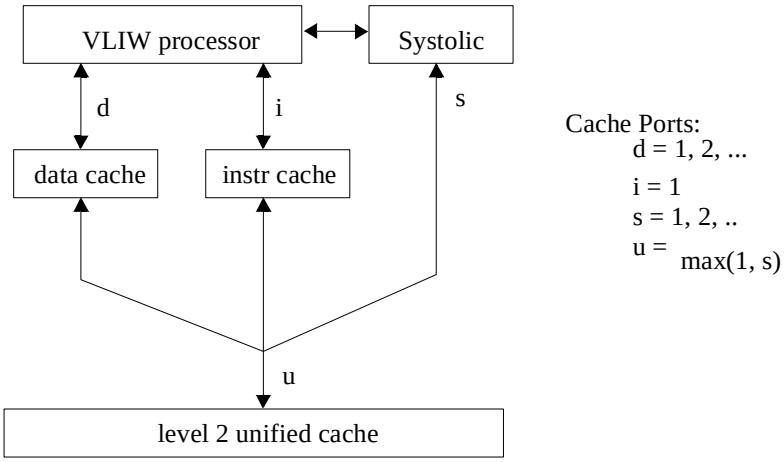


FIGURE 1. Configuration of embedded computer system to be automatically designed. If both the VLIW processor and systolic array are present, the systolic array is controlled by the VLIW. The goal is to automatically design each of the components in this figure along with the interconnect.

The goal of this research is to develop techniques for automating the design of such a system, making the appropriate trade-offs between cost and performance to produce a system that would be competitive with one produced by a human designer. There are at least two anticipated benefits to this automation:

- Thoroughness of exploration. The complex interactions between the components of such a system will force a human designer to rely on judgement and experience in designing them, possibly eliminating interesting configurations, and making “tuning” of the system, for either cost or performance, difficult.
- Time-to-market. Automating the system design might deliver a product more quickly, freeing up engineers for other tasks.

There are three aspects of automated design: (1) design space exploration (called “spacewalking” in this paper) to determine the “optimal” system configuration; (2) synthesis of design points (e.g. to the gate or transistor level); and (3) evaluation of design points to determine their performance. This paper covers only “spacewalking”—evaluation and synthesis issues are being investigated by other members of the Compiler and Architecture Research Group [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11].

2 Spacewalker Structure

Spacewalker is a program that explores a large architectural space, attempting to create a set of "good" architectures for a given application. "Goodness" implies intelligent trade-offs between cost of implementation of the architecture in silicon, and the performance of the application running on that implementation. Its primary input is the embedded application written in C, and its primary output is a set of pareto-optimal systems for that application. In the course of walking the design space, spacewalker relies upon a number

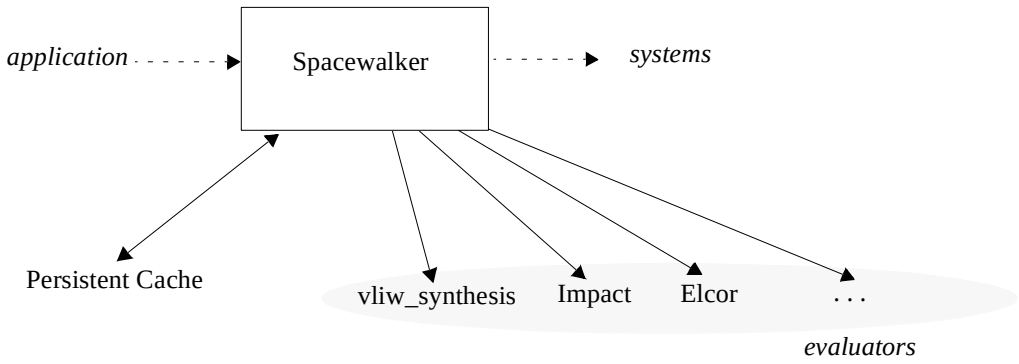


FIGURE 2. Spacewalker invokes several evaluators for creating and evaluating system configurations and saves the result in a persistent cache.

of other programs known as *evaluators (figure 2)*:

- *vliw_synthesis*. Accepts a high-level description of a VLIW processor and generates (1) a target model for the Elcor compiler; and (2) a VHDL description of the hardware to implement that processor.
- *impact*. Used to perform the initial passes of compilation, and also to simulate execution of the application to derive execution statistics.
- *elcor*. A parametric compiler which maps the output of the Impact compiler onto a VLIW model (generated by vliw_synthesis).
- *instruction format optimizer.* Custom crafts an instruction format for a compiled application to reduce code size. When this is used, vliw_synthesis must be re-invoked to create the appropriate instruction decode logic for the VLIW.
- *eas*. Assembler.
- *eld*. Elcor linker; links separately compiled procedures and computes code size.
- *memory_synthesis*. Synthesizes models of the caches.
- *Cheetah simulator.* Estimates effect of caches on system performance.
- *systolic_preprocess* and *systolic_compile*. Transforms systolicizable C code into a hardwired systolic array implementing the same functionality.
- *systolic_synthesis*. Generates the VHDL description of the array created by systolic_compile, along with performance and cost statistics.

Results from the evaluators are stored in a persistent cache on disk.

3 Design Space Parameters

The architectural space explored by spacewalker is defined by the target application and by a set of parameters, which are subdivided into VLIW parameters, memory parameters and systolic parameters.

3.1 Application

Applications must be written in C. If the application contains code that might be mapped onto a systolic array, that code must be separated out into a distinct function (extraction of systolicizable loop nests has not been automated). Only one systolicizable code fragment, and hence only one systolic array, can be implemented for a given application, although this could be extended in the future.

3.2 VLIW parameters

VLIW machines are specified by the following parameters:

1. *Predication*. Either supported by both hardware and compiler or by neither.
2. *Speculation*. Either supported by both hardware and compiler or by neither.
3. *Registers*. Spacewalker may specify the number of registers in each of the register files: gpr (integer register file; minimum registers = 16), fpr (floating point; min = 12), pr (predicate; min = 6), pred (predicate register file, currently fixed at 256 registers because of compiler limitations), and btr (branch target registers; min = 3).
4. *Function Units*. Four different types of function units are available: integer, floating point, memory and branch. In so-called *homogeneous* machines, each instance of a function unit is fully functional and identical to all other instances of that type on a given VLIW machine. In *heterogeneous* machines, each function unit instance of a given type is custom crafted to contain only a subset of all possible operations associated with that type. For example, an application requiring no floating point divides, infrequent floating point multiplies, and frequent floating point adds might be mapped onto a heterogeneous VLIW with two floating point function units where one unit supports multiplication and addition, the second supports only addition, and neither supports division at all. Heterogeneous machines allow for cheaper hardware to be built, tuned for a given application. The algorithm used for creating heterogeneous machines will be covered later when VLIW walking is discussed.
5. *Literal Widths*. Spacewalker may specify the widths of wide literals for each of three types: memory literals, branch literals, and integer data literals.

The following parameters were also exposed by the compiler and synthesis software, but time did not permit exploring them:

1. *Compound Memory Ops*. Spacewalker may explore the trade-offs between supporting only simple loads (with latency n) vs. compound loads (with latency n+1). A given system will support only simple loads or only compound loads, never both.
2. *Exposed Branch Latency.* To compare instruction units with low latency and slow clock with i-units with longer latency and faster clock. It is assumed in this case that arithmetic and memory latencies are independent of clock.
3. *Opcode Emulation*. Allows the compiler to emulate certain opcodes in order to trade-off code size and execution time for simpler hardware.

3.3 Cache Parameters

Each of the three caches, level 1 data cache, level 1 instruction cache, and level 2 unified cache, are independently specified with the following parameters:

1. *Ports*. (must be 1 for instruction cache).
2. *Size*. (must be a power of 2)
3. *Line_size*. (must be a power of 2)
4. *Associativity.*

The instruction cache and second level unified cache also require a fifth parameter known as "dilation" which is used to improve the estimates of cache performance. This will be discussed later in the section on memory walking.

3.4 Systolic Parameters

The following parameters (along with the application's systolicizable nested loop code) specify a systolic array:

1. Shape. Limited to 1 and 2 dimensional arrays. The number of processors in each dimension must be equal to $2^i3^j5^k$ for non-negative integers i, j, k.

2. Bandwidth. The maximum allowable bandwidth between the systolic array and the second level cache, in units of words / cycle.

3. Mapping direction. This may vary from 1 to the loop nest depth of the nested loop executed on the systolic array.

4 Spacewalking Strategy

The goal of spacewalking is to generate custom embedded systems that jointly minimize both cost and execution time of the target application. The designer of an embedded system may have an upper bound on cost and/or an upper bound on the execution time—if a cost upper bound exists, the designer may want the fastest machine that does not exceed that cost; if an execution time upper bound exists, the designer probably wants the cheapest machine that meets that execution time bound. Spacewalker produces a set of pareto-optimal machines and defers the final cost / execution-time trade-off to the user.

Spacewalking is done using a divide-and-conquer approach: the VLIW space, systolic space, and memory space are walked independently, and the results of each walk are then combined to produce a final system walk (only combinations of pareto-optimal subsystems need be considered). Although in principle it would seem more efficient to walk the spaces concurrently, (for example, as coroutines that fed each other useful information to help prune the search space), in practice there does not appear to be much that can be gained by doing that. The reason? VLIW walking dominates computation time by a large margin, so there is little to be gained by pruning the search spaces of memory and systolic. On the other hand, walking the memory and systolic spaces first could help prune the VLIW walk when, for example, there is a known user constraint on either cost or performance for the system, but this has not been explored.

The system spacewalking steps are basically as follows:

1. Do a walk of the systolic design space if the application contains a systolicizable loop nest.

2. Do a walk of the memory design space. This involves doing walks of the data cache, instruction cache, and second level unified cache, and then composing them to produce a set of pareto-optimal memory systems.
3. Do a walk of the VLIW design space.
4. Do a system ‘walk” by composing the results of the systolic, VLIW and memory walks, and doing further refinements of the resulting systems.

5 Systolic Walk

If systolic source code for the application is present and systolic searching is enabled, spacewalker will do an exhaustive walk over the parameters *shape*, *mapping_direction* and *bandwidth*. Evaluating a single systolic configuration requires running *systolic_preprocess*, *systolic_compile* and *systolic_synthesis* to produce the cost and performance data. The results of this walk are used to create *s* Paretos, where *s* is the set of all legal values of memory ports that the systolic array has to the second level cache. Figure 4 shows an example pareto for an application implementing a digital filter.

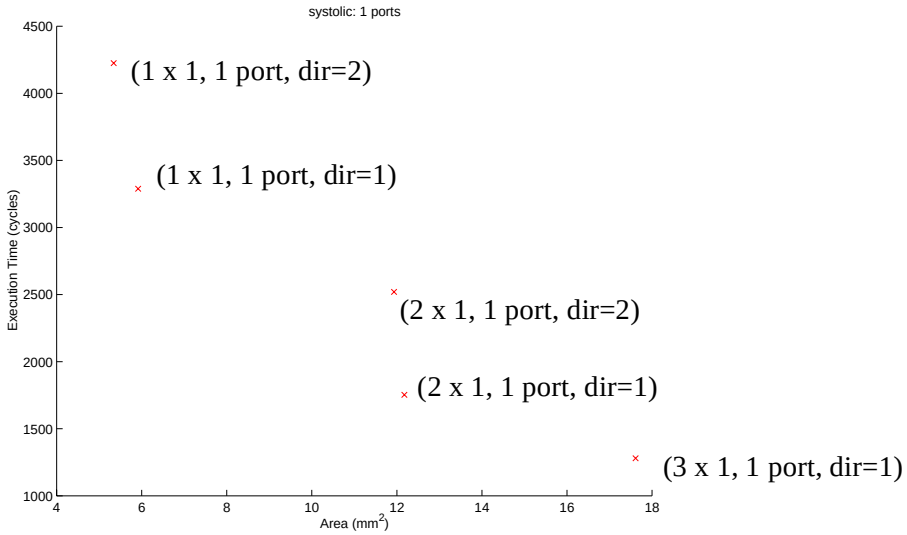


FIGURE 4. Pareto from a systolic walk. The design space (walked exhaustively) included all arrays containing up to three processors, 1 port to the second level unified cache, and all possible mapping directions. Systolic arrays evaluated but eclipsed by the pareto arrays are not shown.

6 Memory Walk

The memory system is broken into five components: RAM, ROM, first level data cache (dcache), first level instruction cache (icache), and second level unified cache (ucache). RAM and ROM are ignored during the memory walk for the following reasons:

- ROM—we assume ROM contains the executable image of the application. The size of this image is a function of the application and the VLIW processor, so the area required for this is computed during the VLIW walk. In the context of spacewalking, ROM is more logically a part of the VLIW than a part of the memory system.

- RAM—this holds the data for the application. We assume that the size of the data area is independent of architecture (may not be true for VLIW vs. VLIW/systolic) and therefore ignore it.

This leaves the three caches to be explored. The model used by spacewalker is shown in the following figure:

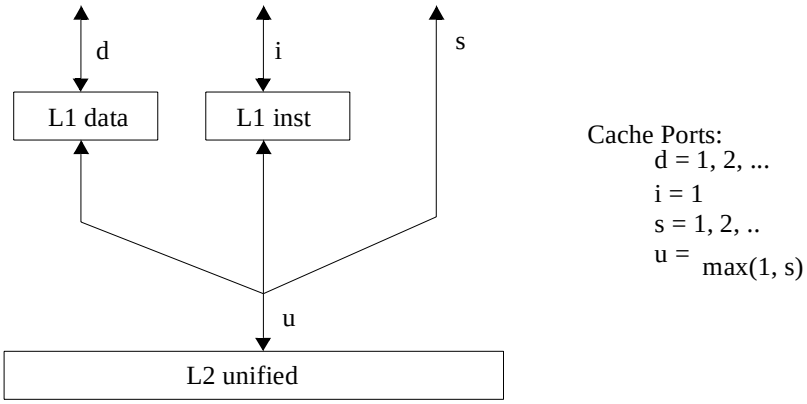


FIGURE 5. Memory system model

The memory system walk involves the following steps:

1. Do an exhaustive walk over the dcache design space, attempting to jointly minimize cache area and cache misses that an application would incur in accessing that cache. (The caches are walked exhaustively due to the design of the Cheetah cache simulator. Cache simulation requires the generation of an execution trace followed by processing of that trace in the simulated cache. Trace generation consumes much more time than cache simulation, so Cheetah generates the trace only once and then simulates the entire cache design space in parallel.) A critical assumption in doing cache evaluations in this walk is that the cache access pattern by the VLIW, and hence the behavior of the cache, is independent of the VLIW design. This walk will produce a set of paretos, one pareto for each possible value of the cache ports parameters (figure 6).
2. Do an exhaustive walk over the icache design space, again jointly minimizing cache area and misses. The modeling of this cache is more complex than it is for the dcache, since the cache behavior is clearly dependent on the VLIW which accesses it: for example, a VLIW which requires a wider instruction word will require more bandwidth from that cache. This effect is indexed with an additional cache parameters called “dilation” which is defined to be the code size of the application on the candidate VLIW divided by the code size on a reference VLIW, and then quantized to a fixed quantization scale. The dilation is needed by the Cheetah simulator to fold in the effect of the VLIW instruction format when estimating cache performance, without having to do a separate instruction cache simulation per VLIW design. This walk produces a set of paretos, indexed by the quantized dilation.
3. Do an exhaustive walk over the ucache design space. Since this cache is accessed by the dcache, icache and systolic array, this walk will produce a three-dimensional set of paretos indexed by the number of dcache ports, the number of systolic ports, and the quantized dilation.

4. Create a three-dimensional set of memory system paretos indexed by dcache ports, ucache ports, and quantized dilation. A (dcache, icache, ucache) tuple may be composed if they share common dcache posts, icache ports and dilation indices. Figure 6 illustrates this composition.

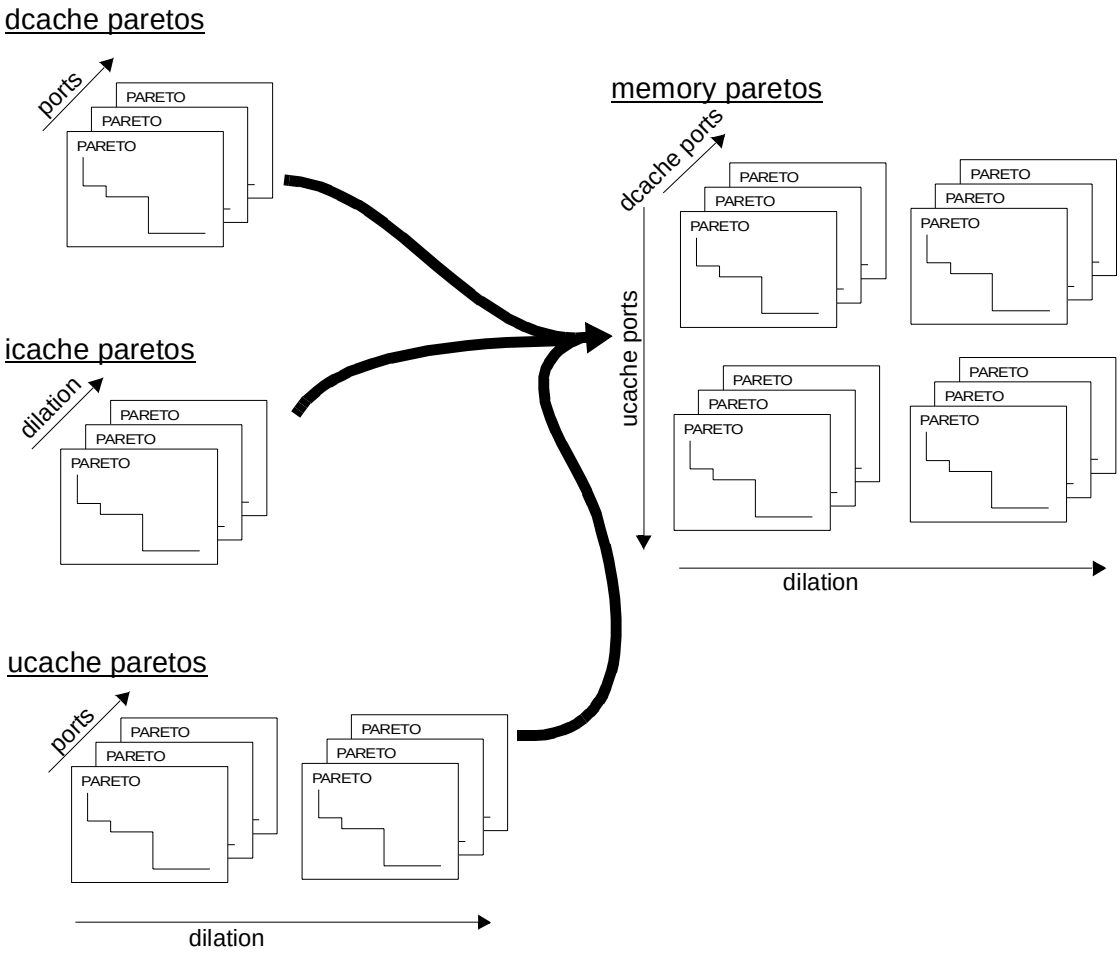


FIGURE 6. Compatible subsystems from dcache paretos, icache paretos and ucache paretos are composed to create sets of memory paretos.

7 VLIW Walk

All VLIW walks involve specifying and synthesizing different VLIW machines and then evaluating the performance of the application on those machines through compilation. The performance estimation is orders of magnitude more expensive in terms of CPU time than the VLIW specification, synthesis and cost evaluation, so it becomes important to limit the number of machines whose performance is evaluated.

7.1 Application Characterization

Before beginning a VLIW walk, the application is first characterized by performing the following steps:

1. A reference VLIW is constructed and the application is compiled onto it. This supplies the reference code size needed for determining the dilation parameters used in modeling the icache and ucache.
2. A histogram of all the literal values in the program is built. This histogram is later used to help optimize the instruction format design.
3. The dynamic and static opcode usage is measured. This is useful later in constructing heterogeneous function units.
4. A table is constructed of (frequency, critical path length) data for each exit from a hyperblock within the program. This table can be used during the walk to estimate the performance of machines that have not yet been evaluated.
5. The application is partially compiled and simulated using the Impact compiler to produce an intermediate representation (IR) annotated with execution statistics. The transformations performed by Impact are independent of the VLIW parameters, except for predication and speculation. This phase is performed only four times for an application, once for each combination of (predication, speculation).

7.2 VLIW Specification and Synthesis

Specification of a homogeneous VLIW requires only specification of the parameters mentioned in section 3.2:

- predication
- speculation
- integer units
- floating point units
- memory units
- branch units
- integer registers
- floating point registers
- predicate registers
- branch registers

Heterogeneous VLIWs require more detailed specification of each instance of a function unit. In the heterogeneous space, a function unit of a given type (say, floating point) consists of one or more opgroups, where an opgroup implements a subset of the operations possible for that type. For example, one floating point opgroup might implement add and subtract, another might implement multiply.

Constructing a set of function units for a heterogeneous VLIW relies upon the dynamic opcode statistics generated by the application characterization step. The following algorithm was used to add a function unit of a given type (such as a floating unit) to a VLIW:

```
addFunctionUnit(vliw) {
    for each opgroup
        tally instances of that opgroup in existing function units in the vliw
    find opgroup with largest neediness
    threshold = largest_neediness * .75
```

```
        create an empty function unit
        for each opgroup type
            if neediness(opgroup type) > threshold
                add opgroup to function unit
        add function unit to vliw
    }

    neediness(opgroup) {
        if dynamic_usage(opgroup) > 0   &&   instances(opgroup) == 0
            return infinity
        else
            return dynamic_usage(opgroup)  /  (instances(opgroup) + 1)
    }
```

Note that if an application never uses any of the functionality within an opgroup, that opgroup never gets instantiated, otherwise the number of instances of an opgroup is roughly proportional to dynamic usage of operations implemented by that opgroup. This simple algorithm offers many opportunities for improvement.

7.3 VLIW Evaluation

The performance of an application on a VLIW can be evaluated by compiling it onto the VLIW and then simulating its execution. This is done in several phases:

- Phase 1: The desired VLIW architecture is specified and synthesized with an unoptimized instruction format.
- Phase 2: The appropriate IR produced by the application characterization step (depending on the desired values of predication and speculation) is compiled with the Elcor compiler onto the desired VLIW target architecture.
- Phase 3: An optional phase (enabled or disabled by the spacewalker user) that takes the output of phase 2, creates an optimized instruction format, and resynthesizes the VLIW so that the correct instruction decode logic will be synthesized. This generally produces a VLIW that consumes more area than the original, but this is counterbalanced by the fact that the new VLIW requires less code size (and hence less ROM area) because of the optimized encoding.
- Phase 4: The compiled application is assembled and linked to determine the application's code size (and ROM area) as well as an estimate of the number of cycles needed to execute it (assuming perfect caches that never miss).

Since Elcor is a research compiler, its execution time is orders of magnitude longer than a production compiler, and phases 2, 3 and 4 are very expensive in terms of CPU cycles (phase 1 is relatively fast). It's possible to replace phases 2, 3 and 4 with a much faster estimation phase that produces a rougher estimate of the number of cycles needed to execute the application. This is done by invoking the Elcor compiler with a special flag that causes it to generate a table of the resource bound path lengths (rbpl) for each hyperblock exit in the program. Performance is then estimated from this table along with the (frequency, critical path length) table built during application characterization using the following algorithm (called the RECMII/RESMII estimator):

```
estimated_cycles = 0
for each hyperblock exit in the application
    estimated_cycles += frequency(exit) * max(critical_path_length(exit), rbpl(exit))
```

7.4 Vliw walking heuristics

With the preliminaries out of the way, the walk begins. A core assumption driving the design of the walk is that evaluating the cost of a machine (VLIW architecture) takes much less computation than evaluating the performance of an application on that machine—as a result, cost is treated as though it comes free, and the word “evaluation” is used in the remainder of this section to mean the evaluation of the performance of an application on a machine.

The following subsections describe the walking heuristics that were explored. In these descriptions, the following terms are used:

- *k*-neighbor. A *k*-neighbor of a VLIW machine is another VLIW machine that has at least 1, and up to k, parameters that are incrementally larger than in the first machine. For example, if machine A has one more integer function unit than machine B, and the two machines are otherwise identical, then A is a 1-neighbor of B (it is also a 2-neighbor, 3-neighbor, ...). For register files, “incrementally larger” is with respect to a quantum not necessarily equal to 1. For example, if the quantum for integer register files were 8, machine A would be a 1-neighbor of machine B if it had 8 more integer registers than B and the machines were otherwise identical.
- -*k*-neighbor. A machine that has at least 1, and up to k, parameters that are incrementally smaller than another machine.
- candidates. A set of unevaluated VLIW machines that will be evaluated during the course of a walk.

7.4.1 Pareto Descent

The pareto descent walk attempts to stay close to the pareto by confining its exploration to the neighborhoods of known, so far, pareto points:

```
pareto-descent(k) {
        candidates += cheapest vliw
        loop {
                remove cheapest from candidates
                evaluate it
                if vliw is a pareto point
                        candidates += k-neighbors of vliw
        } until candidates is empty
}
```

7.4.2 Parallel Pareto Descent

Basically this the same as Pareto Descent modified to exploit parallelism. Spacewalker can distribute multiple machine evaluations across a network and have them execute concurrently, greatly speeding up the walk.

```
parallel-pareto-descent(k) {
        candidates += cheapest vliw
        loop {
                remove all candidates and evaluate in parallel
                for each evaluated vliw
                        if vliw is a pareto point
                                candidates += k-neighbors of vliw
        } until candidates is empty
```

```
}
```

7.4.3 Delft

The Delft heuristic takes multiple sweeps across the design space, from cheap machines to expensive machines and back again, putting more emphasis on minimizing cost or performance, depending on the direction of the sweep. In the following pseudocode, when attempting to find a neighbor with better quality, neighbors are evaluated one at a time, in no particular order, until one with better quality if found:

```
delft-walk() {
        current = most expensive vliw
        for (exponent = 1;  exponent <= 3;  exponent += 0.5) {
                do until current == NULL   // reduce sweep
                        current = -1-neighbor with better reduce-quality
                do until current == NULL   // extend sweep
                        current = 1-neighbor with better extend-quality
        }
}

reduce-quality(vliw) {
        return 1 / (cost(vliw) * cycles(vliw)^exponent)
}

extend-quality(vliw) {
        return 1 / (cost(vliw)^exponent * cycles(vliw))
}
```

7.4.4 Parallel Delft

This is the Delft walk modified to exploit parallelism:

```
delft-walk() {
        current = most expensive vliw
        for (exponent = 1;  exponent <= 3;  exponent += 0.5) {
                do until current == NULL   // reduce sweep
                        evaluate all -n neighbors in parallel
                        current = best -1-neighbor with better reduce-quality, if any
                do until current == NULL   // extend sweep
                        evaluate all 1-neighbors in parallel
                        current = best 1-neighbor with better extend-quality
        }
}
```

7.4.5 Estimated Parallel Pareto Descent

A modification of Parallel Pareto Descent. Instead of evaluating all machines, the performance of some machines is estimated using the RECMII/RESMII estimator when appropriate. It is considered appropriate only when a machine to be estimated is a 1-neighbor of a previously evaluated machine, and differs from that machine only in terms of the number of integer, floating point or memory function units.

```
estimated-parallel-pareto-descent(k) {
        candidates += cheapest vliw
```

```
    loop {
        remove all candidates and  estimate (if possible) or evaluate in parallel
        for each estimated vliw
            if  performance was estimated and bad-machine(vliw)
                discard vliw
            else if vliw is a pareto point
                candidates += k-neighbors of vliw
    } until candidates is empty

badMachine(vliw) {
    vliw2 = -1-neighbor(vliw) that has 1 less function unit of some type
    if estimated_cycles(vliw2) <= estimated_cycles(vliw)
        return true;
    else
        return false
}
```

7.4.6 Conjugate Gradient

Conjugate Gradient is a heuristic for minimizing continuous functions, and has been adapted here to attempt to minimize a discrete function.

```
conjugate-gradient-walk() {
    candidate = cheapest vliw
    gradient = ▽f(candidate)
    conjugateGradient = gradient
    loop {
        compute  ▽f(candidate) in parallel
        update conjugateGradient
        candidate = bestMachine along conjugateGradient vector
    } until hit local minimum
}
```

Two different functions, f, were explored:

- f(vliw) = cycles(vliw)
- f(vliw) = cost(vliw) * cycles(vliw)

7.4.7 Estimated Conjugate Gradient

This is a modified conjugate gradient where RECMII/RESMII estimates are used instead of actual evaluations where feasible when computing the “partial derivatives” of the gradient.

8 System Walking

System walking involves composing subsystem paretos from the subsystem walks and doing additional evaluation of the costs of gluing the subsystems together. Pareto composition is done by taking the “cross product” of compatible Pareto machines (where compatibility is determined by the cache porting parameters, s and d, and by the dilation parameter) to construct a system Pareto. Points in the System Pareto are constructed by adding the costs and cycles of the subsystems that comprise them (figure 7). Once this

Pareto has been constructed, the costs and performances of each point are corrected by the following two operations:

- Cost is recomputed by performing synthesis on each complete machine. This allows us to finally compute the cost of interconnecting the subsystems.

- The cycles needed for the systolic interface code on the VLIW is computed.

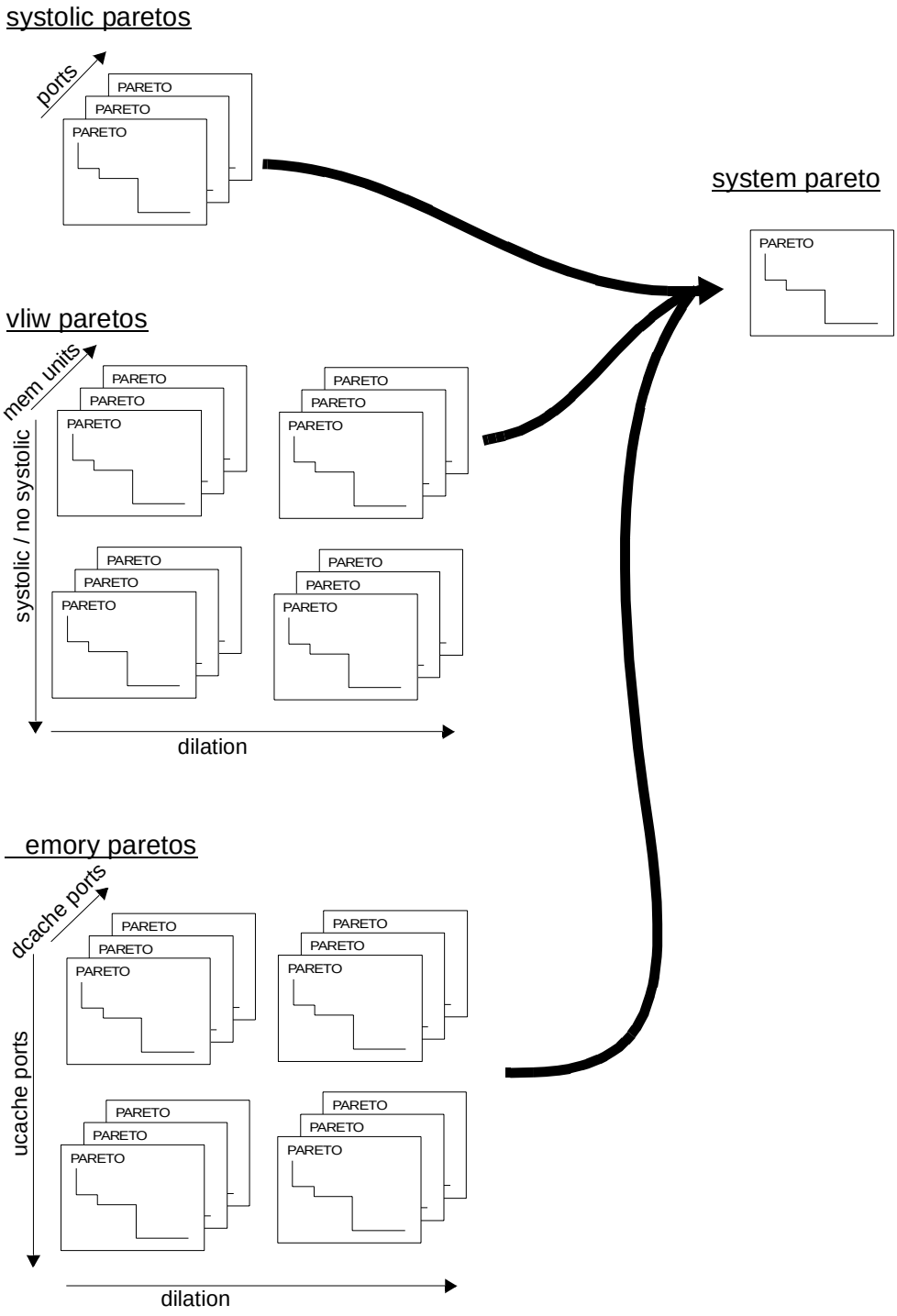


FIGURE 7. Compatible memory, vliw and systolic subsystems are composed to create the overall system pareto.

Additional refinement of the VLIW components was planned but not implemented. These refinements were to include:

- Additional macrocell elimination to reduce register file porting.
- Exclusion based port minimization—adding additional exclusions to reduce porting.
- Exclusion-based instruction format design tuning—adding exclusions to reduce instruction width.
- Additional consideration of opcode emulation.
- Trade-offs between simple and compound memory ops.

9 Experiments

Since the systolic and cache design spaces are walked exhaustively, VLIW walking is the chief walking component to be evaluated. The approach taken was an empirical comparison of several walking heuristics on two very simple applications, strcpy (which initializes a 1024 character array and copies it to another array) and fir_ij (which implements a finite impulse response filter using a window size of 32 on vectors of length 128). More complex applications would have been desirable, but the long compilation times for the current implementation of the compiler make this infeasible (for example, compiling strcpy requires about a minute of CPU time on an HP 700 series workstation, while fir_ij requires about 10 minutes).

For each application, a pseudo-exhaustive walk was executed to provide a baseline set of evaluations for comparing the different heuristics (an exhaustive walk would be preferred but was infeasible because of the size of the design space and the long evaluation times). For strcpy, a 3-neighbor pareto descent walk was used; for fir_ij, a 2-neighbor pareto descent was used. It's difficult to know apriori what a good "pseudo-exhaustive" walk would be, but the k-neighbor walks seemed intuitively appealing, and it was found at the conclusion of the experiments that there was only one pareto point found by any of the heuristics that was not also found by the baseline walk, so this decision was probably not unreasonable. The baseline was modified before analysis to include the union of all walks. Of course there may be interesting areas of the design space that were unexplored by any of the heuristics, but short of a true exhaustive walk there's no way of knowing for sure.

The primary goal of the experiments was to compare the quality and efficiency of the different walk heuristics. Efficiency was measured in terms of wall clock time. Comparing quality was much more difficult because it's not clear what an objective measure of that might be. For this report, the quality of a walk, Q, was defined to be:

$$Q = \text{area(walked pareto)} / \text{area(baseline pareto)}$$

although many other reasonable metrics are possible.

9.1 Design Space Limits

The actual design space explored for each application is shown in the following table:

Design Space for Experiments

parameter	**strcpy**	**fir_ij**
predication	0, 1	0, 1
speculation	0, 1	0, 1
integer function units	1, 2, 3, 4, 5, 6, 7, 8	1, 2, 3
floating pt. function units	1, 2, 3, 4, 5	1, 2, 3, 4, 5
memory units	1, 2, 3, 4	1, 2
integer registers	16, 24, 32, 40, 48, 56, 64	16, 24, 32, 40, 48, 56, 64
floating pt. registers	16, 24, 32, 40, 48, 56, 64	16, 24, 32, 40, 48, 56, 64
predicate registers	256	256
branch registers	8, 12, 16	8, 12, 16
total machines:	94,080	17,640

9.2 Walk Quality

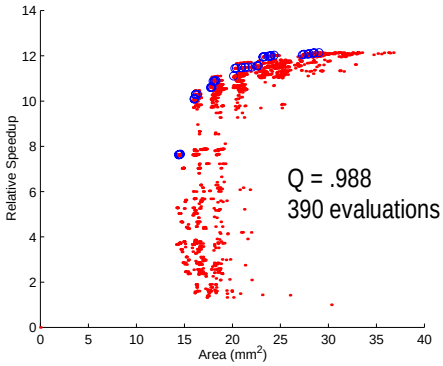
The results of executing all of the walks on fir_ij are shown in figure 8. In each plot, the red dots represent the union of all experiments done—the baseline—and the blue circles represent the pareto points found by a given walk. Each figure also shows the quality, Q, of that walk, as well as the number of machines that were evaluated during the walk. The number of evaluations does not translate to wall clock time, however, because some of the walks executed evaluations in parallel across a network.

The highest quality walks was the Parallel Pareto Descent (Q = .999) of 1-neighbors. It was anticipated that a 1-neighbor walk might miss some pareto points because of complex interactions between the VLIW parameters (e.g. increasing the number of integer function units might not help unless the number of integer registers were increased as well). However, the 1-neighbor walk did not miss any pareto points found by the 2-neighbor walk (the same was true for the strcpy walk, not shown).

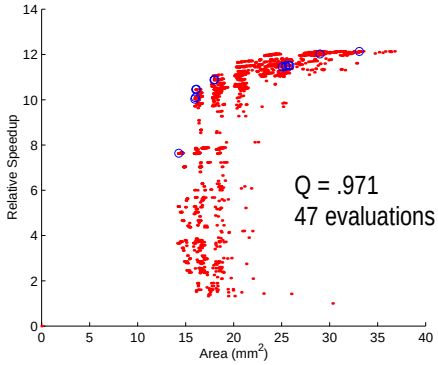
Using the RESMII/RECMII estimator reduced the number of evaluations and walk time (compare the 204 evaluations of the Parallel Estimated Pareto Descent with the 467 evaluations of the Parallel Pareto Descent) but the quality of the walks using the estimator was considerably lower.

FIGURE 8. VLIW walks

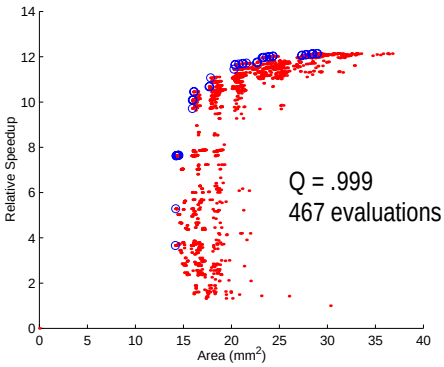
Pareto Descent (1-neighbors)



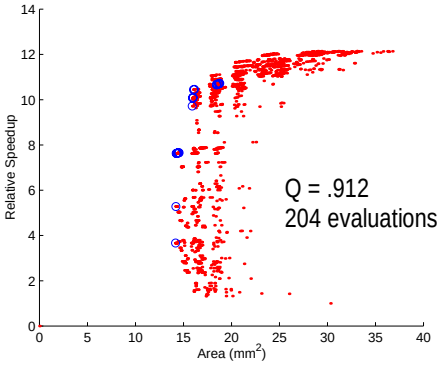
Delft



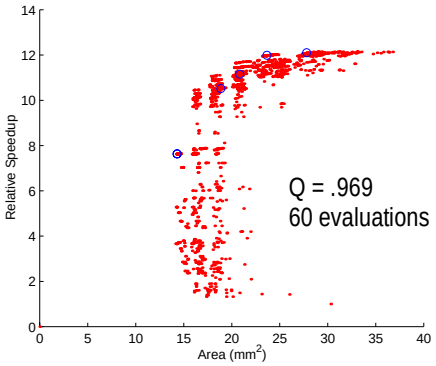
Parallel Pareto Descent (1-neighbors)



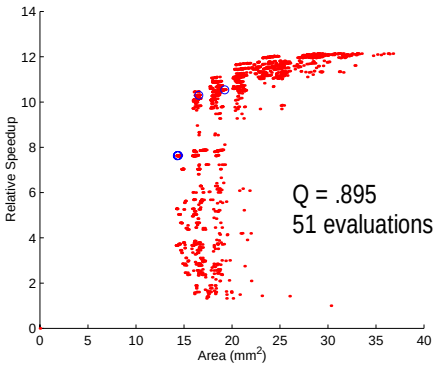
Parallel Estimated Pareto Descent (1-n)



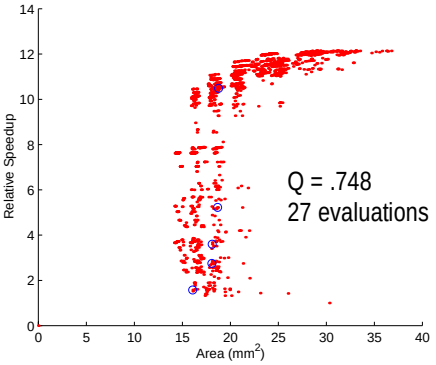
Conjugate Gradient (min cycles)



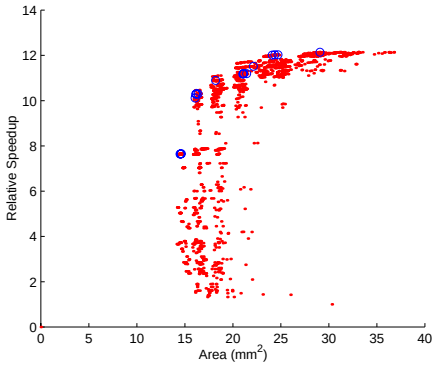
Conjugate Gradient (min cycles * cost)



Est. Conjugate Gradient (min cycles)



Parallel Delft



9.3 Walk Efficiency

Walk time spanned multiple days for some of the heuristics, during which processor load varied. In order to correct for this somewhat, a model was build that considered the maximum amount of evaluation parallelism available at every step in the walk, and (with the assumption that all evaluations required the same amount of processing time) computed the relative wall clock time each heuristic would require as a function of the number of CPUs available for use across a network (figure 9).

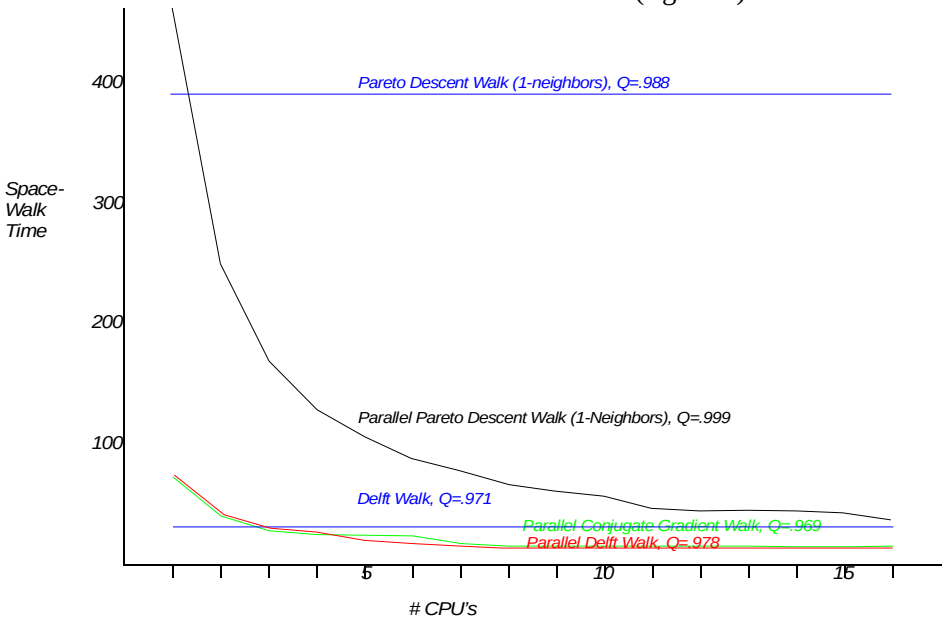


FIGURE 9. Relative execution times of walks on fir_ij.

9.4 Heterogeneous vs. Homogeneous Function Units

Heterogeneous machines were expected to offer better cost / performance ratios, and this was the case (figure 10). Heterogeneous function units cut the cost sometimes more than half for a given performance level. The highest achievable performance of the heterogeneous machines did not approach that of the homogeneous (12x vs. a 17x speedup) but this is probably to a large degree due to the very simple heterogeneous construction algorithm used.

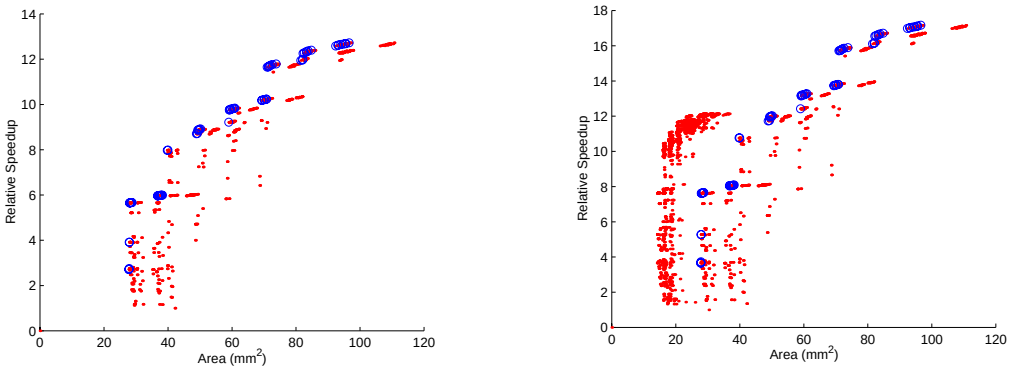


FIGURE 10. Comparison of VLIW walks with heterogeneous versus homogeneous function units. Heterogeneous function units offer reduced cost for a given performance level.

9.5 Compiler Monotonicity

In principle, if the Elcor compiler is provided two VLIW target architectures, one a superset of the other, on which to compile an application, the performance of the application on the more powerful architecture should be no worse (and hopefully better) than on the weaker. But since compilers solve NP-complete problems and cannot find optimal solutions, they will sometimes fail to achieve this ideal. Figure 11 shows a study the monotonicity of performance as a function of the number of resources, assuming a fixed clock rate, for the Elcor compiler based on all of the strcpy and fir_ij experiments.

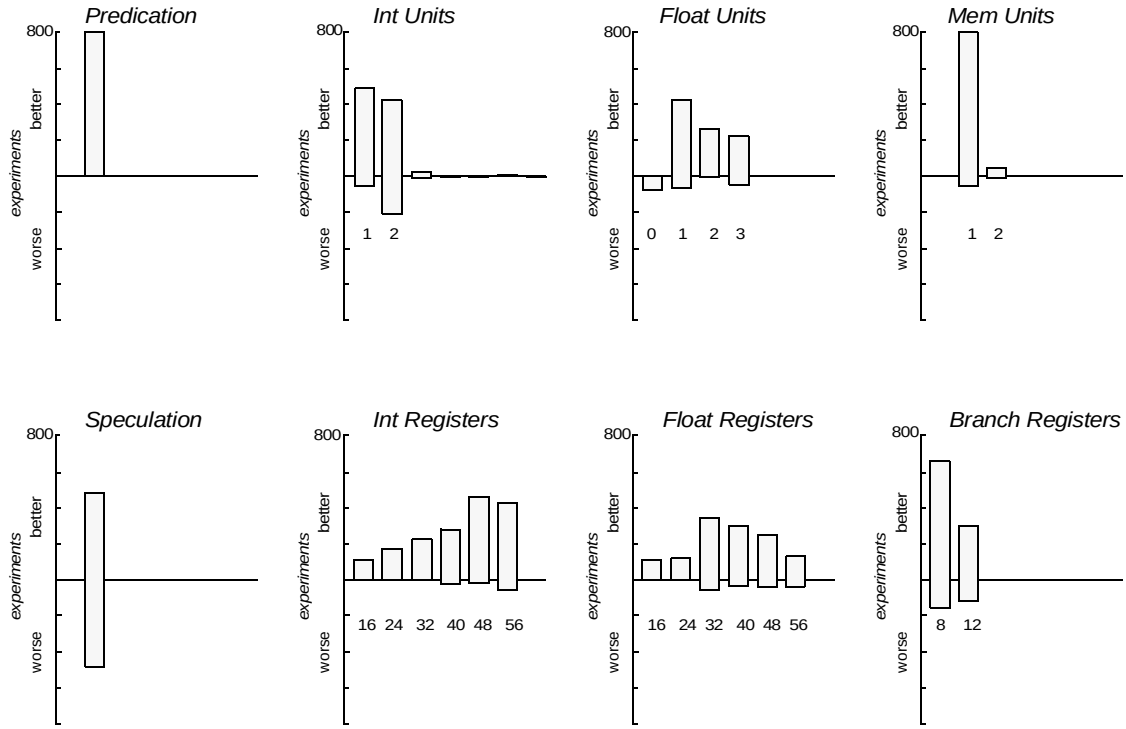


FIGURE 11. Monotonicity of performance as a function of resources for the Elcor compiler. The effect of increasing VLIW resources on the performance of the application, assuming constant clock speed. Each plot shows the number of experiments on fir_ij and strcpy where incrementally increasing a resource caused the estimated performance to improve or stay constant (the part of the bar above the x-axis) and the number of experiments where performance declined (the part of the bar below the x-axis). The small numbers below the bars in the 6 right-most plots show the baseline. For example, the left-most bar in the "Int Units" plot shows that when the number of heterogeneous integer units was increased from 1 to 2, there were about 500 experiments where performance stayed the same or got better, and about 50 experiments where performance got worse. With an ideal compiler, performance should never get worse because of the addition of more functionality.

9.6 Cost Sensitivity

Figure 12 shows the percent increase in the cost of a VLIW when incrementally increasing each of its resources for the fir_ij application. Figure 13 shows the same plots when considering the cost of both the VLIW *and* the ROM needed to hold the application's code. These show how supplying additional registers

can, in the right situations, increase the VLIW cost but reduce the total cost by reducing code size (presumably through reduced register spill).

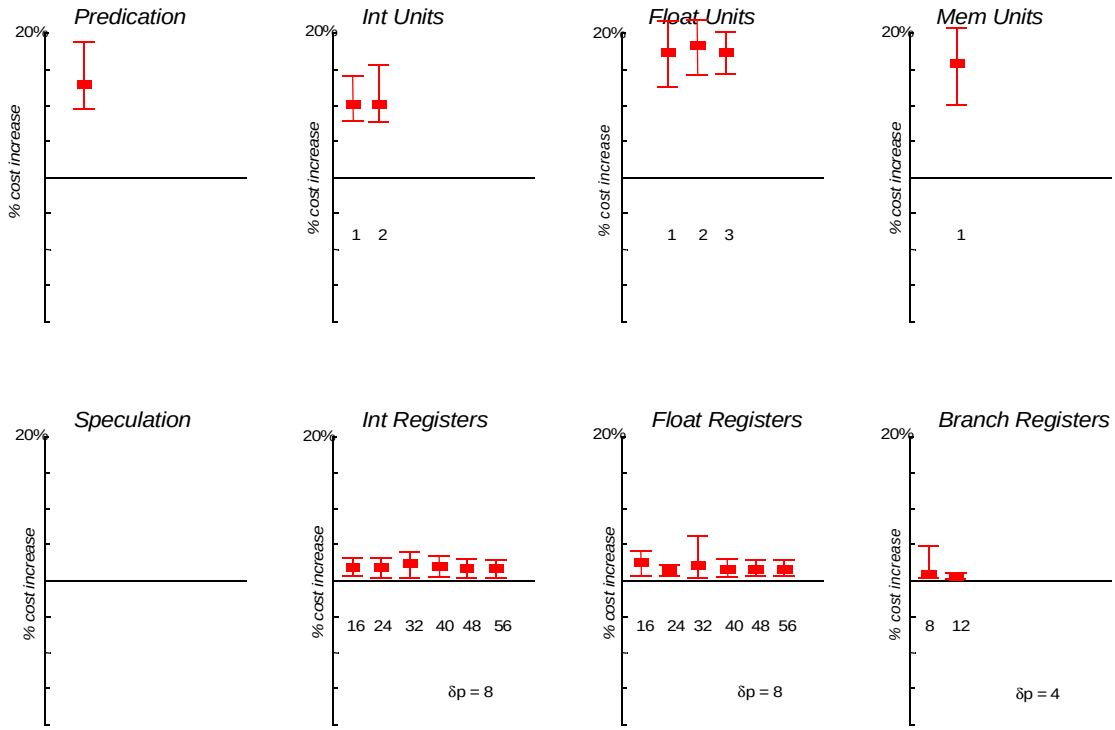


FIGURE 12. The effect of increasing VLIW resources on the cost of the VLIW for the fir_ij application. Each plot shows the (min, mean, max) percent increase in the VLIW cost when incrementally increasing a resource. The small numbers below the bars in the 6 right-most plots show the baseline. For example, the left-most bar in the "Int Units" plot shows when the number of (heterogeneous) integer function units was increased from 1 to 2, this resulted in a a 10% increase in VLIW cost when averaged over all experiments.

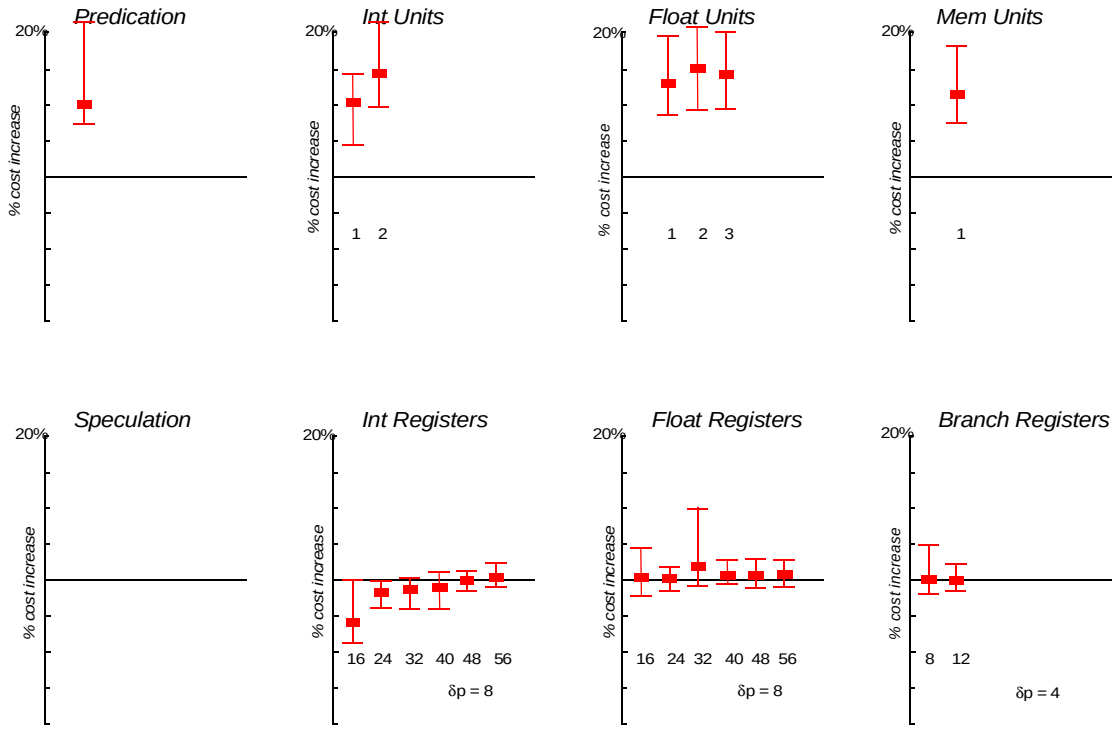


FIGURE 13. The effect of increasing VLIW resources on the cost of the VLIW + ROM for the fir_ij application. Compare with figure 12.

9.7 Performance Sensitivity

Increasing the number of VLIW resources should speed up (or at least not slow down) an application, assuming a fixed clock speed. Figure 14 shows the effect of incrementally adding resources on the performance of fir_ij. Note the diminishing returns for this application as resources continue to be added.

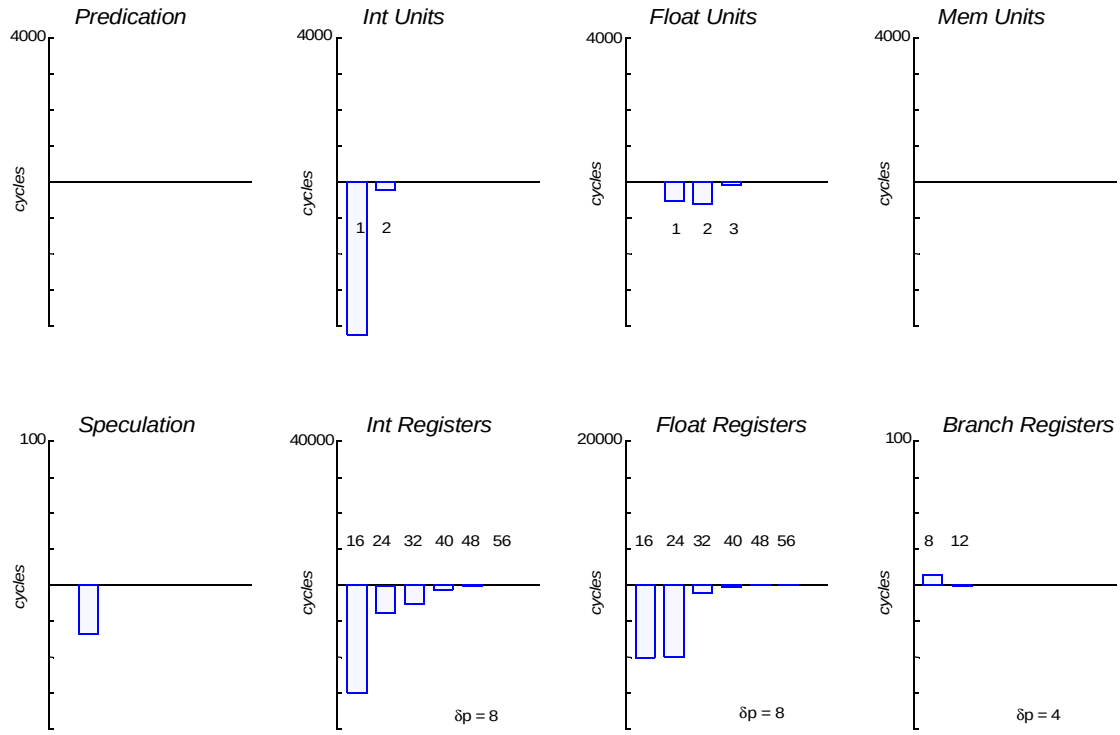


FIGURE 14. The effect of increasing VLIW resources on the performance of fir_ij. Each plot shows the change in the number of cycles needed to execute the application as a result of incrementally increasing a resource (negative changes are desirable). The small numbers below the bars in the 6 right-most plots show the baseline. For example, the left-most bar in the "Int Units" plot shows when the number of (heterogeneous) integer function units was increased from 1 to 2, this resulted in a reduction of 4000 cycles for executing the application.

10 Conclusions

The experiments conducted were, for pragmatic reasons, conducted on only two, very small applications, so making extrapolations is difficult. Only the following conclusions are offered:

- Parallelism can be effectively exploited in spacewalking by distributing evaluations across a network of processors.
- Heterogeneous function units offer significantly reduced cost for a given performance level. There are likely further optimizations possible from improving the heterogeneous function unit construction algorithm.
- The RECMII / RESMII estimator could not be successfully exploited in these experiments.
- Selection of an appropriate walking policy depends upon (1) the desired quality (Q) of the walk; (2) the calendar time available for performing the walk; and (3) the number of networked computers

available for executing the walk. Figure 15 shows the quality / execution time trade-offs for a network of ten processors. In this scenario, Parallel Delft would be the policy of choice if walking time needed to be minimized, while Parallel Pareto Descent would be preferred if the maximum walk quality were desired.

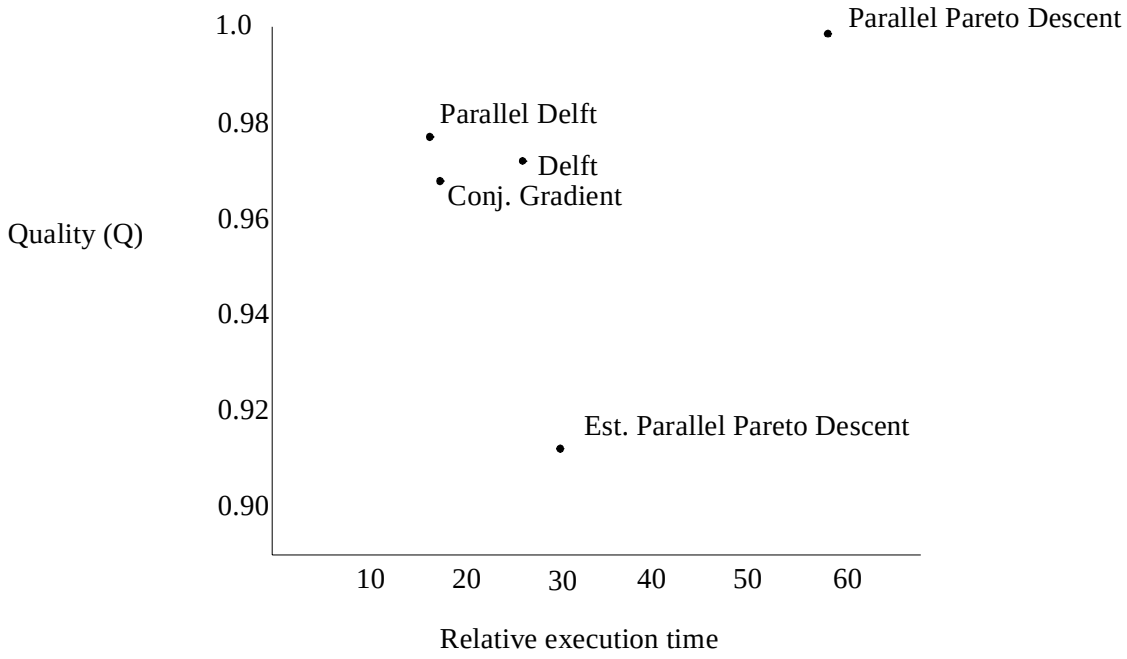


FIGURE 15. Quality / Execution time trade-offs for several of the walking policies.

11 References

[1] Schreiber, Robert; Aditya, Shail; Rau, B. Ramakrishna; Kathail, Vinod; Mahlke, Scott; Abraham, Santosh; Snider, Greg; "High-Level Synthesis of Nonprogrammable Hardware Accelerators," HP Labs Technical Report HPL-2000-31

[2] Abraham, Santosh; Rau, B. Ramakrishna; Schreiber, Robert; "Fast Design Space Exploration Through Validity and Quality Filtering of Subsystem Designs," HP Labs Technical Report HPL-2000-98

[3] Aditya, Shail; Mahlke, Scott A.; Rau, B. Ramakrishna; "Code Size Minimization and Retargetable Assembly for Custom EPIC and VLIW Instruction Formats," HP Labs Technical Report HPL-2000-141

[4] Rau, B. Ramakrishna; Schlansker, Michael; "Embedded Computing: New Directions in Architecture and Automation," HP Labs Technical Report HPL-2000-115

[5] Aditya, Shail; Mahlke, Scott A.; Rau, B. Ramakrishna; "Code Size Minimization and Retargetable Assembly for Custom EPIC and VLIW Instruction Formats," HP Labs Technical Report HPL-2000-141

[6] Aditya, Shail; Rau, B. Ramakrishna; "Automatic Architecture Synthesis and Compiler Retargeting for VLIW and EPIC Processor," HP Labs Technical Report HPL-1999-93

[7] Abraham, Santosh; Rau, B. Ramakrishna; “Predicting Load Latencies Using Cache Profiling,” . HP Labs Technical Report HPL-94-110

[8] Rau, Bantwal Ramakrishna; Kathail, Vinod; Aditya, Shail; “Machine-Description Driven Compilers for EPIC Processors,” HP Labs Technical Report HPL-98-40

[9] Schreiber, Robert; Rau, B. Ramakrishna; Darte, Alain; Vivien, Frederic; “A Constructive Solution to the Juggling Problem in Processor Array Synthesis,” HP Labs Technical Report HPL-2000-30

[10] Aditya, Shail; Kathail, Vinod; Rau, B. Ramakrishna; “Elcor's Machine Description System: Version 3.0,” HP Labs Technical Report HPL-98-128

[11] Schlansker, Michael; Rau, B. Ramakrishna; “EPIC: An Architecture for Instruction-Level Parallel Processors,” HP Labs Technical Report HPL-1999-111