Reconfigurable Memory Queues/Computing Units Architecture

Akhilesh Tyagi
Department of Computer Science
Atanasoff Hall
ISU, Ames
IA 50011-1041

submitted to: National Science Foundation

Contents

1 Project Summary

Memory bandwidth has been growing at a much slower rate than processing bandwidth. The main technique to close this gap between memory bandwidth and computing rate (bandwidth) is to provide larger on-chip storage. Larger on-chip storage (registers and caches) reduces the memory bandwidth needs due to locality exhibited by typical programs. It has recently been demonstrated that hardware controlled on-chip storage is not able to exploit the locality to the best extent and hence results in larger than necessary memory bandwidth requirements.

This proposal introduces another technique to ameliorate the memory bandwidth problem. A typical processor already has several deeply pipelined computing units – such as multipliers, division units, and ALUs. When the processor is waiting on data/instructions due to insufficient memory bandwidth, many of these computing units are idling. The proposal is to dynamically reconfigure the idling computing units into on-chip FIFO (queue) storage. This reconfigurability comes at a very minimal hardware cost in area and cycle time overhead. The reconfiguration can be done under compiler control. By turning some of the idling computing units into queue memory, the computing rate goes down reducing the number of memory accesses proportionately. Additionally, the resulting enlarged on-chip storage also helps the memory bandwidth providing the compiler with a 'double-edged' sword. This helps to run the processor in a 'balanced' mode most of the time, where computing rates are balanced with the memory access rates (neither computing resources, nor memory resources are underutilized or starved).

An architecture with reconfigurable memory/computing units needs a compiler that can use FIFO access pattern of a queue effectively. The known register allocation techniques work only for a random-access on-chip storage. We propose several compiler algorithms for queue-allocation.

2 Project Description

2.1 Introduction

The Problem: Each problem when computed at certain computational rate (instructions per second) results in certain memory bandwidth requirements (bytes per second). The memory bandwidth needs depend on the size and organization of the on-chip memory (registers and caches). If the memory bandwidth of the processor exactly matches the memory bandwidth requirements of a computation with a fixed local memory configuration, the computation is said to be *balanced*[*]. Note that a processor implementation fixes all the three parameters of interest in this context: computation rate, on-chip (local) memory configuration and memory bandwidth. For a fixed processor (or fixed computation rate and local memory configuration), different problems achieve this balance for different values for memory bandwidth. Some problems underutilize the available memory bandwidth, some may be balanced, and some stall often when the physical memory bandwidth is not sufficient. It would be a futile proposition to design a processor with fixed configurations for all the three attributes – computation rate, local memory and memory bandwidth, that would be balanced for most general purpose computations. However, if these implementation parameters were dynamically configurable we may be able to move the processor configuration towards balanced (not necessarily balanced) for the problem at hand.

The Proposed Solution: The proposed approach makes computation bandwidth dynamically tradeable with local memory size. This trade-off is best accomplished with hardware units that can either act as an efficient computational unit (such as an adder, a multiplier, and a division unit) or as an efficient random-access memory. Unfortunately, we have not been able to design such a hardware unit. However, most modern processors (particularly superscalar processors) already contain deeply pipelined function units which can also be used as local memory. One additional multiplexor per pipeline stage will allow the data either to bypass the combinational logic in a pipeline stage or to go through it for normal computation. In the first case, a k-deep pipeline function unit behaves as a k entry FIFO store (or LIFO, stack store). The second option uses it as a function unit as originally intended.

We propose that the dynamic configuration be performed under compiler control. Which address space would this additional local storage be embedded in? The two options are as follows. It could be transparent and fit into the machine address space like a cache. However, the hardware would have to manage the dynamic configuration in this case. We believe this to be a suboptimal configuration for reasons similar to the ones found in [BGK96]. The second option is to provide a compiler-managed address space for these queues (such as registers). We intend to investigate this model. As an example, consider a hypothetical superscalar implementation with 6 deeply pipelined units: 2, 4-stage FP adders; 2, 8-stage multipliers; and 2, 16-stage division and complex arithmetic units. The dynamic store address space in this case is: $A0, A1, M0, M1, D0, D1$ with capacity of 4, 4, 8, 8, 16, 16 words respectively. Any instruction that could use a register argument can now use a queue argument, *e.g.*, `ADD R5, M1, A0`. A reference to a queue argument also advances the queue head. Similarly, a load or store can use a queue argument, *e.g.*, `LW A1, 55(R1)` would queue the contents of memory location given by `55(R1)` into the queue `A1`. We may also need explicit queue advancing instruction to provide the compiler with more generality, `ADVANCE M1`. Explicit configuration instructions include `FUNCTION`

[*]Kung [Kun86] introduced this term.

`M1` to convert unit `M1` into a function unit, `MEMORY M1` to force `M1` into queue memory configuration, and `TOGGLE M1` to toggle `M1`'s mode between function unit and memory modes.

A computation often stalled for memory access due to lack of memory bandwidth could convert some of the computation units into memory units. This has two desirable effects – *reduced computation rate* resulting in a lower demand on memory bandwidth and *increased local memory* also potentially reducing memory bandwidth requirements. Alternately, a computation underutilizing the memory bandwidth with some computation units configured in memory mode could convert some of the local queue memory into function units. This may result in a higher computation rate without overwhelming the memory bandwidth.

If a compiler controls the configuration of all the reconfigurable function units, it needs very good compile-time models to predict the state of a computation: underwhelmed or overwhelmed memory bandwidth. An alternative or supplemental method could be hardware assisted reconfiguration. The implementation could include a *memory bandwidth flag* which is set to *true* when the memory bandwidth is overwhelmed for certain fixed or compiler controlled time window. The implementation could detect this condition by keeping track of memory stalls count. The compiler could then insert instructions to reconfigure some of the function units into memory mode *conditioned* upon this flag.

Another advantage of queue memory scheme (or any scheme that can access multiple data items for each decoded address) is lower energy per local memory access. Of course, an effective use of such an implicit addressing for local data depends critically on the compiler's ability to schedule these storage queues. This is reminiscent of some of the early computing machines that used mercury delay loops for memory. Systolic algorithms [Kun79], [Mol83] solve a similar problem of synchronizing the availability of data with its use. Reif and Tyagi [RT91] use optical delay line loops for optical computing facing similar problems at algorithm design level. Some compiler level transformations (particularly, array blocking [LRW91]) for scientific data would be directly applicable with block size equal to the queue size. Compiler would have to develop long enough traces of *definitions* and *uses* to fill these queues for data arising in non-numeric applications.

Deliverables: The preceding discussion devotes many more words to the description of the reconfigurable memory architecture than to the compiler techniques needed to assign variables to queue memories. However, it is the compiler technology development that is likely to take more effort than the development of the architecture. The key deliverable from this research would be the compiler technology to use non-random-access memories for local storage. This technology would be applicable even in a non-reconfigurable architecture which supports fixed length queues of registers (every register address refers to a queue of registers rather than a single register). This type of *implicit addressing* may be beneficial for low power, portable computing applications. The second deliverable would be an assessment of reconfigurable memory architectures. This assessment is to be carried out through simulations. The hypothesis is that reconfigurable computing/memory architectures alleviate the memory bandwidth problem present in the current architectures. The degree of reduction in memory bandwidth needs would be determined by the proposed research.

Organization: The next section provides an overview of the related work. The architecture and implementation concerns are presented in Section 2.3. Section 2.4 discusses some compiler strategies to employ these queues gainfully. Section 2.5 concludes the proposal.

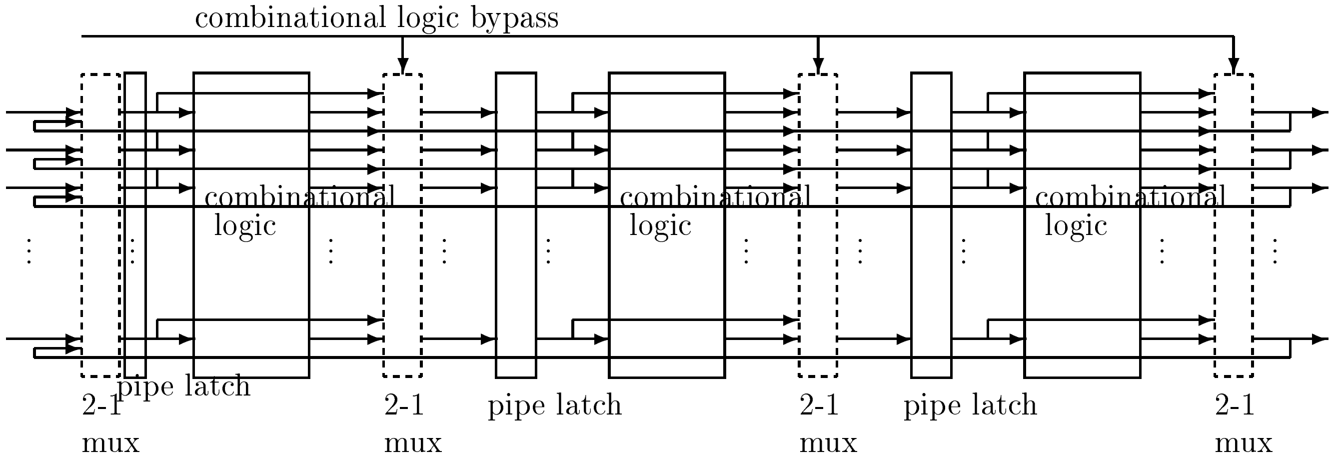


Figure 1: A Reconfigurable Pipelined Unit

2.2 Related Work

The notion of reconfigurability in computer architecture has been around for a long time. Estrin [Est60] considered adaptive computation through the use of a small set of application specific computation elements in addition to the computation core. Cocke and Miller [CM73] provide for dynamic configurability of interconnections between function units to adapt to the communication requirements of the underlying algorithm. Rauscher and Agrawala [RA78] reconfigure the microcode of the machine to suit the problem characteristics. Athanas and Silverman [AS93] describe a dynamically generated FPGA based implementation through high-level co-synthesis of a program. Virtual Wires [BTA93] speeds up logic emulation through FPGA reconfigurability reducing inter-chip pin bandwidth.

Kung [Kun86] was the first theoretical study of the processing rate and memory bandwidth balance. This paper defines a *balanced* implementation with respect to a problem P to be one where the I/O bandwidth equals the computation bandwidth. The question addressed then is how much should the I/O bandwidth be increased to keep the implementation balanced with respect to P if the computation bandwidth is increased by a factor α. Kung demonstrates lower bounds on the local memory size. M_{old} is the original local memory size and M_{new} is the required local memory size for rebalancing the processor: matrix multiplication: $M_{new} \geq \alpha^2 M_{old}$, FFT and sorting: $M_{new} \geq M_{old}^{\alpha}$. Most of the problem decompositions for these bounds were originally derived in Kung and Hong [HK81]. Several papers [Gan87, GJG88] have discussed blocking of arrays and vectors for exploiting the physical characteristics of the memory hierarchy. For instance, in order to fit the numeric data in a cache block/line, one can create blocks of cache line size rather than the square root of cache size used in Kung's scheme. Lam *et al.* [LRW91] quantify the effects of cache interference on blocking.

Burger, Goodman and Kägi [BGK95], [BGK96] argue that the current local memory management techniques (caches) and latency tolerating techniques are creating imminent memory bandwidth problems. They show that the cache efficiency in terms of fraction of cache data that is live is very low. [BGK96] shows an increase in the memory bandwidth when latency tolerating/reduction techniques such as out-of-order-issue and lockup-free caches are employed. One of their conclusions is to let compiler manage and allocate a larger set of variables. The proposed scheme fits in with their

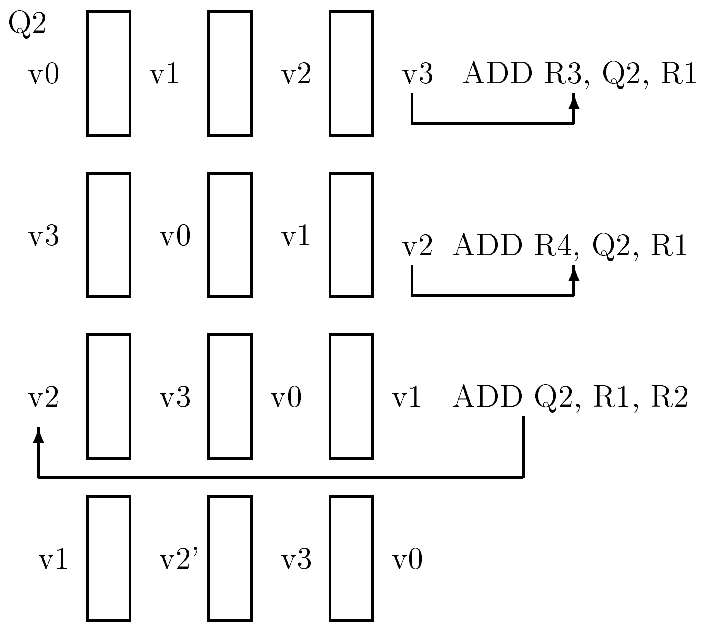


Figure 2: Data Cycling on Queue Access

recommendation.

2.3 Architecture and Implementation Issues

The main architecture level change is the addition of another address space – the processor-chip queues. Several deeply pipelined function units, already available in the processor (particularly superscalar), can be reconfigured to behave as a FIFO (queue) memory. All we need is a bypassing path for the combinational logic in each pipe stage. Figure 1 shows a simple multiplexor based organization for this reconfigurability. A multiplexor potentially adds some delay to each pipe stage.

Note that since the main objective of any local storage is to contain a subset of the working set of a program, the FIFO queue in Figure 1 has been arranged as a *circular* queue. This adds to the routing cost of the implementation of these function units. However, we believe that a circular queue provides a better model for locality containment (and for compiler) than a plain FIFO. A *write* into this queue enters the pipeline at the first pipe stage, the location where arithmetic data for function computation would have normally entered. Similarly, a *read* of queue reads the data at the output pipe stage (the output stage is the default queue *head*). This is to minimize the overhead logic needed to support the reconfigurable function units. This, however, forces all the *reads* of a data to occur at least $k-1$ cycles after its *write* for a k-deep pipeline ($k = 3$ in Figure 1). This restriction could be too limiting if our compiler experiments suggest that most data are read within 2 cycles after their write. In this scenario, we could move the write port to 2 stages before the output stage.

We are assuming that all the function units that are pipelined are designed to be reconfigurable with extra multiplexors as in Figure 1. Given that the extra pipe stage latch cost of making the *initiation interval* of a function unit equal to 1 is also benefiting the local storage in these architectures, all function units may be designed with an initiation interval of 1. In the rest of the proposal, we assume that each function unit is designed with a pipeline depth to support an initiation interval of 1.

Instruction Set: Let the set of reconfigurable function units be {`Q0, Q1, ..., Qk`} with storage capacity (pipeline depth) `n0, n1, ..., nk` respectively. In the following, we assume the instruction format of DLX [HP96], an architecture similar to MIPS R3000, where the destination operand is followed by the source operands. New instructions to control the reconfigurable units include instructions to alter the behavior of the unit:

`MEMORY Q1`: configures unit `Q1` to behave as a FIFO memory.

`FUNCTION Q1`: configures unit `Q1` to behave as a function unit.

`TOGGLE Q1`: toggles the behavior of unit `Q1` between FIFO memory and function unit modes. This instruction may not be implemented for efficiency considerations, but we mention it for the sake of completeness.

Data in these queue memories can be accessed in the same way as the register data is accessed. Hence, one of the function unit names `Q0, Q1, ..., Qk` can replace a register name in any instruction.

`ADD R5, Q2, R1` adds the contents of register `R1` and function unit `Q2`† (the data at the output port of `Q2` when this instruction is executed) and puts the result in `R5`.

`ADD Q1, Q2, R1` adds the contents of register `R1` and queue `Q2` and puts the result in the input stage of queue `Q1`.

`LW Q2, 50(Q0)` loads the contents of memory location given by adding 50 to the contents of value at the head of `Q0` into the input port of `Q2`.

An important point to note here is that an access to a queue `Qi`, either as a source operand or as a destination operand, *implicitly advances the queue by one position.* Hence in the instruction sequence `ADD R3, Q2, R1; ADD R4, Q2, R1` the second `ADD` gets a different value from `Q2` than the first `ADD`. Figure 2 demonstrates this point. The first row shows values v0, v1, v2 and v3 located at pipe stages 1 2, 3, and 4 respectively. The `ADD R3, Q2, R1` instruction receives the value at the queue output stage (stage 4), v3. This `ADD` instruction also circulates the values so that now values v3, v0, v1, v2 are at stages 1, 2, 3, 4 respectively. The second `ADD R4, Q2, R1` instruction gets v2, the value now at the queue head. Again, this `ADD` implicitly advances `Q2` so that v2, v3, v0, v1 are at stages 1, 2, 3, 4 respectively. A write into `Q2` now by instruction `ADD Q2, R1, R2` writes the result (value v2') into the input stage (stage 1) of `Q2` and advances the queue. After this write, values v1, v2', v3, v0 are at stages 1, 2, 3, 4 respectively. Note that these three instructions need not be processed at consecutive cycles for this queue behavior. As long as there are no other instructions accessing `Q2` in between these `ADD` instructions, the values accessed in Figure 2 remain valid. The data in a queue memory stays put during the cycles when no instruction accesses its data.

Do we need instructions to move data between register file and memory queues or between memory queues? Physically, this capability is available just as in a typical RISC processor through the use of the register `R0`. `ADD Qi, Qj, R0` moves the contents of `Qj` to `Qi`. Similarly, `ADD Qi, R5, R0` moves

†provided `Q2` is configured as a memory unit when this instruction is executed, otherwise it is an error.

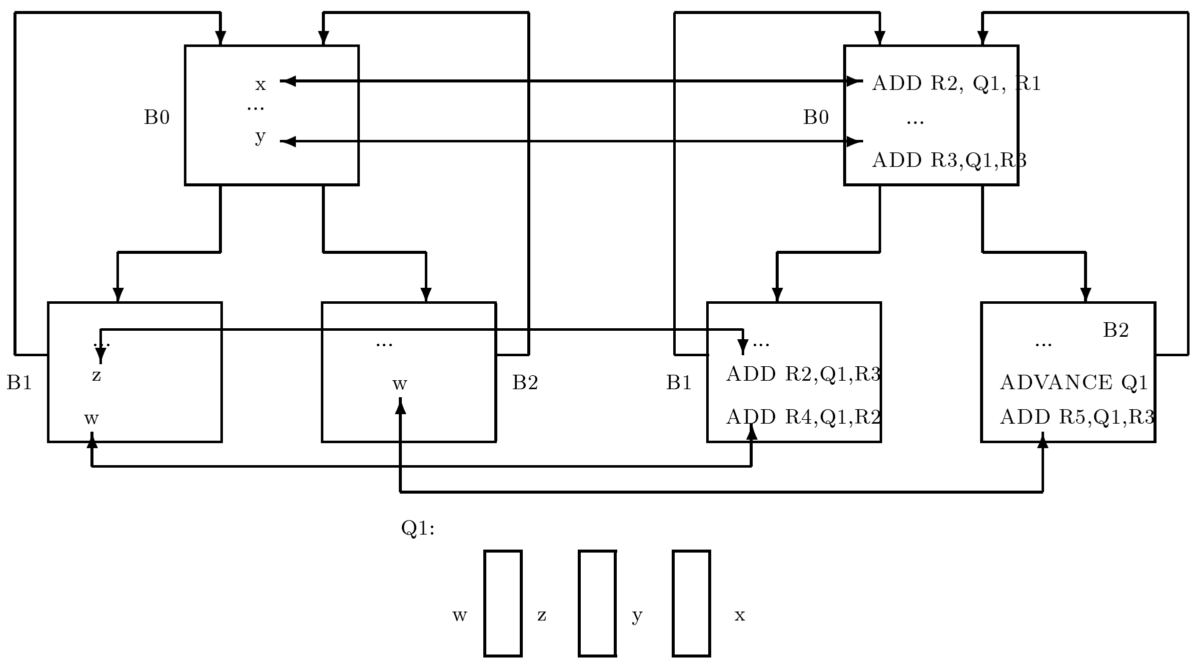


Figure 3: An Example Use of `ADVANCE` instruction

the contents of `R5` to `Qi`. Data movement between register file and memory queues could be exploited by the compiler to keep a subset of the spilled registers in these queues and restore them from there. We discuss the compiler issues in Section 2.4.

There would be times when we would like to circulate the queue entries by one position without accessing the data. Such an instruction provides the compiler and programmer greater flexibility in using the memory queues.

`ADVANCE Q2` advances the data in queue `Q2` by one position. This instruction could be used by the compiler to align the data for some computation path where the value at the queue head is not needed. Consider the example shown in Figure 3. Basic blocks B0, B1 and B2 form a loop. Basic block B0 is followed by either B1 or B2 depending on the outcome of the branch instruction. Basic block B0 accesses the variables x and y in that order. Basic block B1 accesses z and w and B2 accesses only w. Let us assume that all these accesses are read accesses within `ADD` instructions. The compiler can allocate variables x, y, z and w on a 4-deep memory queue `Q1` in the order shown on the bottom of Figure 3. Let us assume that w, z, y and x are in stages 1, 2, 3 and 4 respectively of `Q1` on entry into B0. The basic block DAG on the right in Figure 3 shows the code generated for these blocks. Note that on entry into B2, the queue `Q1` needs to be advanced before being accessed since the variable z is not used in B2.

Instruction Encoding: Our intent is to expand the register field in an instruction to include the memory queues. A DLX or MIPS instruction devotes 5 bits to each register field (example register-register instruction shown in Figure 4). Adding one more bit to the register field (a 6-bit field) gives us capability of accessing up to 32 such memory queues. Most MIPS/DLX instructions do not need

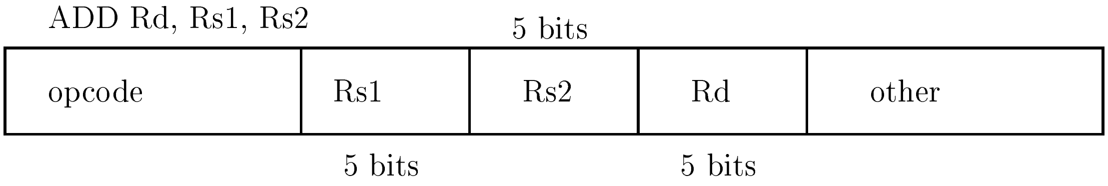


Figure 4: Instruction Encoding in MIPS/DLX

all the 32 bits and hence these extra bits can be accommodated. A *compare and branch* instruction such as `beq` in MIPS would have only 14 bits for the branch offset with this change. One solution is to only support comparison against zero, as in DLX, `beqz`, which fits into 32 bits comfortably despite the 6-bit size for the register field. Other option, of course, is to shorten the branch offset to 14 bits. Yet another possibility is to reduce the number of registers to 16 and use 16 memory queues while retaining 5-bit register field size. If the memory queues are effective, 16 registers might suffice.

Terminology: In the following, we define a few terms pertaining to the use of queues.

Definition 1 *The read-read-latency of a queue is defined to be the minimum number of cycles between the two reads of the same data. It would typically be depth of the queue pipeline.*

For example, the read-read latency of Q2 in Figure 2 is 3. If many data values are read several times in a quick succession, then automatic circulation of data on a read is a bad design decision. In this case, queue data could be supported in two addressing modes – *circulating queue read* and *static queue read*. Circulating queue read is the read behavior shown in Figure 2. In a static queue read, the data stays put in their old locations even after a read, *e.g.*, assuming that `ADD R3, Q2, R1` performs a static read of `Q2`, the queue still retains the data v0, v1, v2 and v3 at pipe stages 1 2, 3, and 4 respectively. Note that a static queue read provides a read-read-latency of 0. An extra bit in the encoding of the queue operand could distinguish between static and circulating accesses. Alternately, opcodes could encode the information about static versus circulating operand access.

Definition 2 *The write-read-latency of a queue is defined as the minimum number of cycles between a write and a following read of the same data.*

The queue design assumed in this section performs a write into the input port of the queue pipeline. The earliest read of this data occurs at the output port after (pipeline depth-1) cycles if it keeps moving one stage per cycle. For example, the write-read-latency of Q2 in Figure 2 is 2. Once again, this might be too limiting for the compiler if most data is read right after its write. In such a case, the write port could be moved to the pipestage preceding the output stage to provide a 0 write-read-latency.

2.4 Compiler Techniques

The availability of memory queues poses a new set of problems for the compiler. The compiler is responsible for *memory queue allocation*. Memory-queue allocation refers to the problem of assigning variables and computed values to the queues. Memory-queue allocation is a harder problem than register allocation due to non-random-access nature of the queues. For register allocation, the variables are first allocated to pseudo-registers or symbolic registers. A symbolic register interference graph is constructed where two vertices have an adge if the variables corresponding to the two vertices could be simultaneously live. A coloring of the interference graph gives a register allocation, where each distinct color corresponds to a physical register. Register allocation is known to be an NP-complete problem [Set75]. Graph coloring is also known to be NP-complete [GJ79]. However, a class of heuristics for coloring a register interference graph, introduced by Chaitin [CAC$^+$81, Cha82], seem to be particularly successful.

The memory queues seem to be appealing as target on-chip storage for structured data in scientific computations. Several strategies have been developed to deal with the memory hierarchy for large arrays and vector data [Gan87, GJG88]. A technique known as blocking [LRW91] can be effective at reducing the memory bandwidth needs of an algorithm. The main idea is to operate on the blocks of data (of size B or $B \times B$) for a carefully chosen block size B. The intent is to have the blocks of data of size B or $B \times B$ fit into a cache line. For instance, the number of memory accesses in the classical matrix-multiplication algorithm to multiply two $N \times N$ matrices can be reduced from $2N^3 + N^2$ to $2N^3/B + N^2$ with blocking. Cache allocated blocks, however, generate self-interference within the cache and hence the memory bandwidth gains may not be as large as expected [LRW91]. A compiler controlled storage would be a better candidate to allocate these data blocks. Queues provide an alternative, compiler controlled storage. We first discuss the allocation of queues in scientific computations followed by the queue allocation problem for non-numeric data.

Queue Allocation for Scientific Programs: Consider the matrix-multiplication algorithm for computing $Z = X * Y$ where Z, X, Y are $N \times N$ matrices:

```
for(i = 0; i < N; i = i + 1)
    for (k = 0; k < N; k = k + 1){
       r = X[i, k];
       for (j = 0; j < N; j = j + 1)
          Z[i, j] = Z[i, j] + r * Y[k, j]; }
```

This algorithm makes $2N^3 + N^2$ memory references (if every reference has to go to the memory). However, the same row of Z (ith) is used across all iterations of j-loop. Similarly, entire matrix Y is used across each iteration of i-loop. Hence if the cache were capable of holding a row of Z and entire matrix Y, there would be significant savings in the number of memory accesses ($3N^2$). It is unrealistic to expect the cache to be able to contain $N + N^2$ entries of a row of Z and entire Y for an arbitrary N. Blocking exploits the same idea taking into account the cache size. Only a subset of the ith row of Z of size B and a $B \times B$ submatrix of Y are needed for the algorithm at a time. Here is the *blocked* version of the matrix-multiplication algorithm:

```
for(kk = 0; kk < N; kk = kk + B)
   for(jj = 0; jj < N; jj = jj + B)
       for(i = 0; i < N; i = i + 1)
           for (k = kk; kk < min(kk + B - 1, N); k = k + 1){
              r = X[i, k];
              for (j = jj; j < min(jj + B - 1, N); j = j + 1)
                 Z[i, j] = Z[i, j] + r * Y[k, j]; }
```

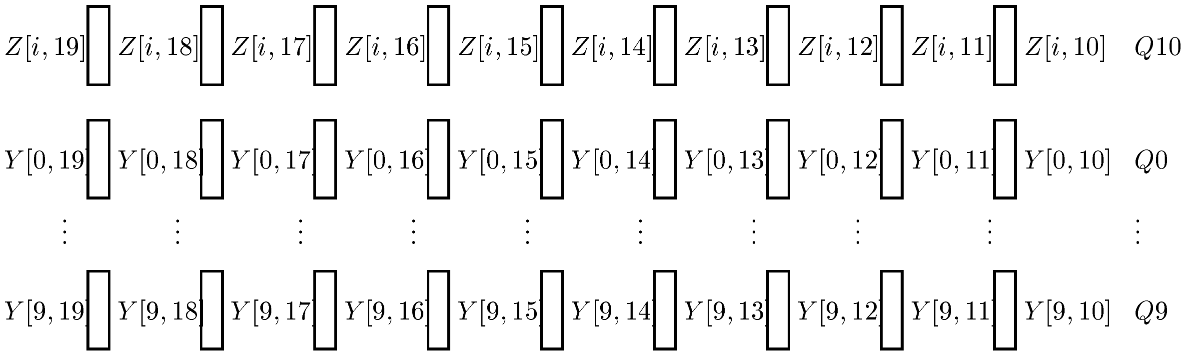


Figure 5: Queue Allocation for Matrix Multiplication

The blocked version makes only $2N^3/B + N^2$ memory accesses assuming that B entry subset of the ith row of Z and a $B \times B$ submatrix of Y can simultaneously fit in the cache. As Lam *et al.* [LRW91] point out, since cache is not explicitly compiler-directed there can be interference between the data blocks themselves. We could use the queues instead to hold the blocks of data. For matrix multiplication, for instance, with $B = 10$, $Q2$ with $n_2 = 10$ could hold $Z[i, 10{*}c]$, $Z[i, 10{*}c{+}1]$, $Z[i, 10{*}c{+}2]$, $\ldots$, $Z[i, 10{*}(c{+}1){-}1]$. Similarly, assuming that there are 10 other queues with at least 10 entries, each row of the $10{\times}10$ submatrix of Y, $Y[k, 10{*}c]$, $Y[k, 10{*}c{+}1]$, $Y[k, 10{*}c{+}2]$, $\ldots$, $Y[k, 10{*}(c{+}1){-}1]$ can be assigned one such queue. This allocation results in a factor of 10 reduction in the memory bandwidth. Figure 5 shows one possible queue-allocation when $j = 1$ and $kk = 0$. We assume the existence of 11 queues: $Q0, Q1, \ldots, Q10$ of depth 10. The Z-row is allocated to $Q10$ and the ith row of the 10×10 submatrix of Y is allocated to Qi for $0 \leq i \leq 9$. This allocation requires the write-read-latency of these queues to be 0, while the queue implementation presented in Section 2.3 has write-read-latency of 8 for $Q0, Q1, \ldots, Q10$. The blocked version of matrix-multiplication requires a read-read-latency of 9, which matches the design parameters of Section 2.3.

Similar blocking schemes exist for several other problems such as FFT and sorting [HK81], [Kun86]. The identification of blocking could be done either in high-level preprocessing of the program, or could be done through compiler techniques reported in Callahan *et al.* [CCK90] and Wolf and Lam [WL91]. Note that allocating arrays to another address space (such as register and queues) is not feasible in a programming language with pointers (such as C). Potential aliasing keeps the compilers from allocating registers for array entries. Assuming that aliasing is not a problem (in a language such as Fortran), several techniques exist to force the allocation of array entries on registers. Callahan, Carr and Kennedy [CCK90] replace array entries by scalar temporaries (called *scalar replacement*) so that register allocation can place them in the registers. Same preprocessing steps (such as scalar replacement) would force the compiler to consider placing the blocked arrays in the queues during its queue allocation phase.

Queue Allocation for Non-numeric Scalar Data: Since register allocation is a very closely related problem, we first present a quick overview of coloring based register allocation heuristics (based on Chaitin [Cha82]). An interference graph $G = (V, E)$ is derived from the flow analysis [ASU86] where $v \in V$ corresponds to a program variable or an intermediate computed value. An edge $(u, v) \in E$ if $u, v \in V$ are simultaneously live. Given k physical registers, the coloring heuristic first looks for any unconstrained vertices (vertices with less than k neighbors) in G. It repeats the following step

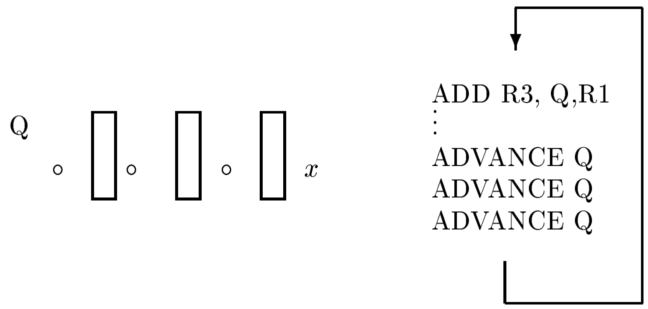


Figure 6: A Queue Allocated Single Periodic Access Variable

until G is empty. If an unconstrained vertex exists, it is deleted from G (since their coloring can be derived trivially given colors of the remaining vertices). Otherwise, it spills a vertex to the memory (added to the spill list). This vertex is chosen heuristically to reduce constraints on other vertices (a vertex of high degree is favored) and to minimize the cost of keeping it in memory. When G is empty, the vertices from the unconstrained vertices list can be colored in the reverse order of their removal. Extra spill code is introduced for the vertices on the spill list. There are many variants on this basic scheme, differing in the following aspects: the definition of live range (from each definition to its last use or entire live range) for building the interference graph; order in which unconstrained vertices are removed; cost function for determining vertices to spill. The nodes to be colored could also be chosen based on some priority [CH84]. Some register allocators perform local allocation (within a basic block) followed by a global allocation phase [CH84], [PF92].

How should these queues be allocated to nonnumeric scalar variables, the class of variables typically allocated on registers? Since the same set of values are candidates for either register allocation or queue allocation, the interaction of the two allocation schemes may be crucial in the quality of the results. One choice is to perform queue allocation on the nodes on the spill lists of typical coloring based register allocation [Cha82], [BGG^{+}89], [BCKT89]. In this model, the most important variables get allocated on the registers and some set of spilled variables is allocated on the queues. However, this may not be the best use of queues. The distinguishing characteristics of two storage classes are: random access for registers and periodic access (based on queue latencies) for queues. Hence the variables exhibiting a very periodic and regular pattern of accesses are best allocated on the queues, while the variables with random access patterns (unpredictable and not periodic) should be assigned to registers. The opposite approach would be to first allocate potentially periodic access variables to the queues. The classical register allocation can then be performed on the remaining variables. It is conceivable that the removal of periodic access variables can remove register pressure considerably resulting in better register allocation. This certainly is likely to be the case with blocked matrix multiplication. In the following, we propose an algorithm to allocate data on the queues. Perhaps, the best method would be to integrate register allocation and queue allocation. The nodes or groups of nodes in an interference graph are simultaneously evaluated for both register allocation and queue evaluation. We propose a variant of Gupta *et al.* coloring [GSS89] method as an integrated allocation methodology.

Queue Allocation: We have stated earlier that the variables with periodic accesses are ideal candidates for queue allocation. This tells only part of the story. If there is only one periodic access variable live in some section of the code, putting it on a queue may not be a good idea due to large read-read

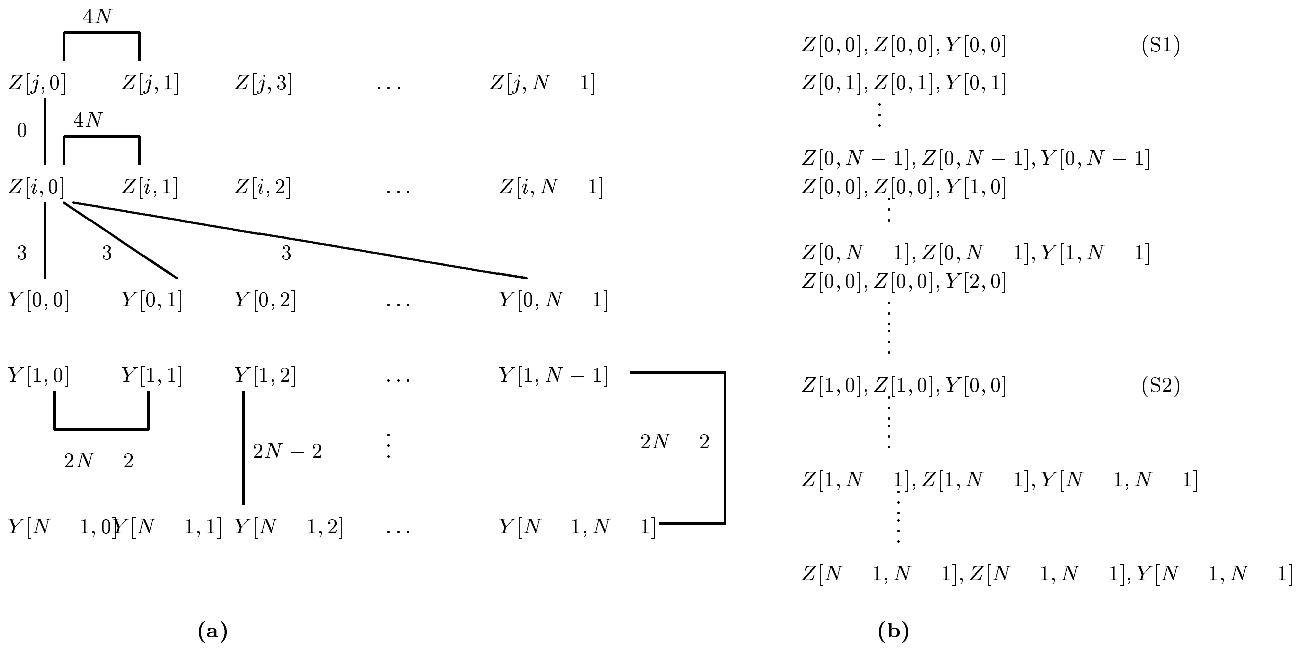
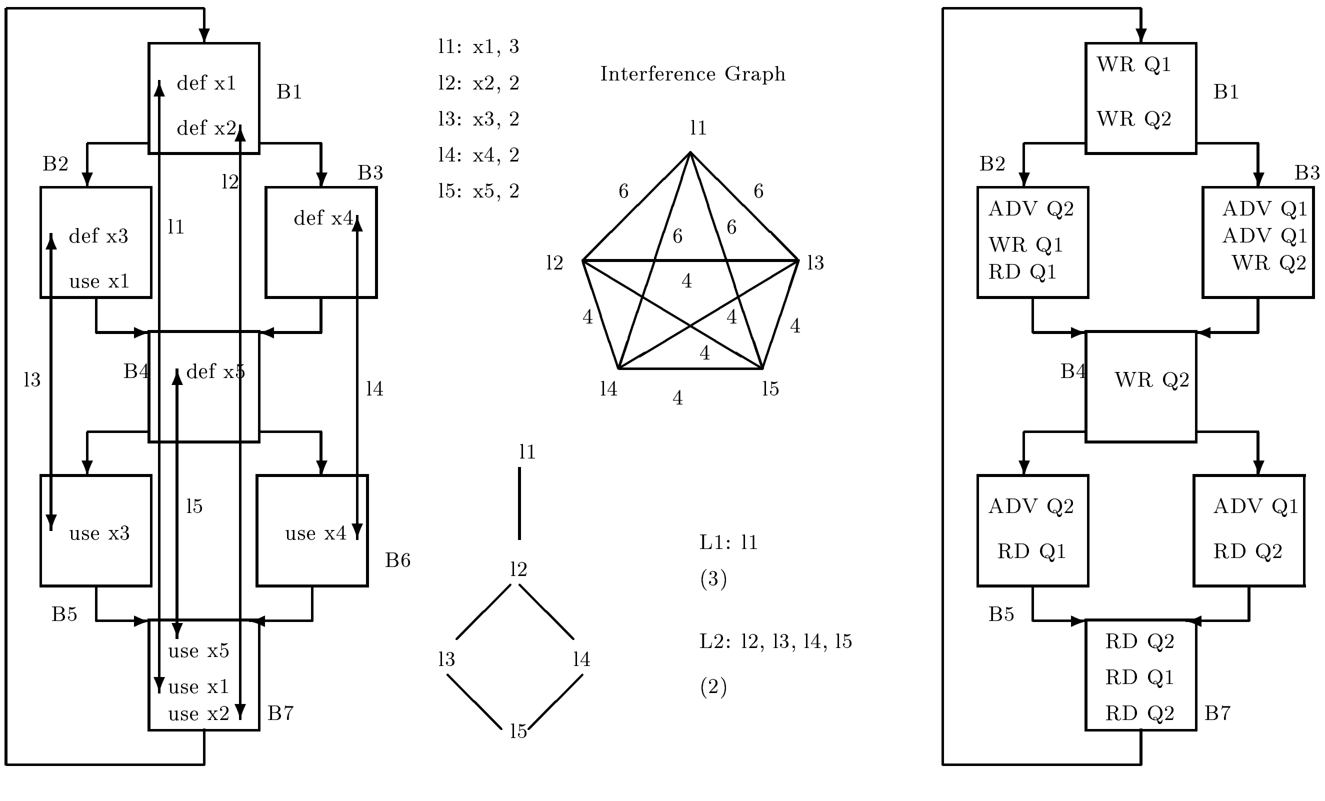


Figure 7: Weight Computation for Interference Graph

latency. We would need to insert several `ADVANCE` instructions between successive reads as shown in Figure 6. The variable x is allocated to the queue Q. Assume that accesses to x occur with a period of 6 cycles (or instructions). If none of the operands of these 6 intervening instructions can be allocated on Q then x would have to be advanced explicitly by insertion of 3 explicit `ADVANCE` instructions. If we support static queue read, then x could stay at the queue head and no `ADVANCE` instructions are needed. However, the queue usage (only one of the four entries in the queue is filled with non-bubble data) is low. Hence, we need to look for a set of variables that are accessed several times periodically within the same code span.

Input to Queue Allocation: We need to identify groups of symbolic registers that are all accessed several times within the same code span. In other words, these variables' live ranges need to interfere. Hence the interference graph $G = (V, E)$ built for register allocation can be used for queue allocation as well. Maximal *cliques* in G correspond to the groups of variables that are simultaneously accessed and hence form good candidates for queue allocation.

An edge between two vertices u, v in the interference graph G only indicates that the live ranges for u and v intersect. We would like the variables in a clique to have many accesses in their live ranges. Hence, a *weighted interference graph* $G = (V, E, W)$ with $W : E \to R$ forms a better input for queue allocation. The weight function W assigns a weight (a real number, but integers might suffice) to each edge $(u, v) \in E$. $W(e = (u, v))$ is a measure of frequency of accesses to variables corresponding to u and v within the intersection of their live ranges. The weight function could be chosen to be a function similar to priorities in [CH84] and [GSS89]. Let us define the *affinity* or weight between two vertices u and v. Let $u \cap v$ be the *intersection* of two live ranges u and v, the code that is common to the two live ranges.



(a) **(b)** **(c)**

Figure 8: An Example Queue Allocation

$$Z[0,0] \longrightarrow Z[0,1] \longrightarrow Z[0,2] \longrightarrow \ldots \longrightarrow Z[0,N-1]$$

L1: $Z[0,0]$ $Z[0,1]$ $Z[0,2]$ $\ldots$ $Z[0,N-1]$
(N)

Q1: $Z[0,0]$ $Z[0,1]$ $Z[0,2]$ $\ldots$ $Z[0,9]$

Figure 9: Queue Allocation for 0th Row of Z

Definition 3 *Let number of definitions and uses of the variable corresponding to u within a live range l be given by $a_l(u)$. Then the weight (or affinity) between the vertices u and v is computed as $W(u,v) = a_{u\cap v}(u) + a_{u\cap v}(v)$. It is the sum of the number of definitions and uses of both variables within the intersection of their live ranges.*

Figure 7 illustrates this weight function. This is an interference graph for the $N \times N$ matrix multiplication algorithm presented earlier. We choose matrix multiplication for this example due to its structured computation and due to the fact that we already know an optimal queue allocation for it (Figure 5). The same ideas apply for nonnumeric data as well. Each node is labeled by the corresponding matrix entry (such as $Z[0,1]$ and $Y[1,3]$). It represents the global live range for the corresponding matrix entry (from the first definition to the last use). Figure 7 (b) shows the access pattern and order for these matrix entries. Note that live range for $Z[0,0]$ is from statement labeled (S1) to statement labeled (S2) (in fact this is the live range for the entire 0th row of Z). Note that $Z[i,k]$ does not intersect with $Z[i',k']$ for $i \neq i'$ and hence the affinity (weight) between two Z vertices from different rows is 0 (or there is no edge). For Z vertices within the same row, $Z[i,j], Z[i,j']$, their live ranges intersect for N^2 entries ((S1) to (S2) for row 0 elements in Figure 7 (b)). There are $2N$ uses and definitions of $Z[i,j]$ and $2N$ uses and definitions of $Z[i,j']$ within this intersection. Hence $W(Z[i,j], Z[i,j']) = 4N$ as shown in Figure 7 (a). Intersection of live range for $Z[i,j]$ and $Y[i',j']$ is again limited to the span of row i of Z. In fact, it is limited to just the statement where $Y[i',j']$ is used within the ith row range. $Z[i,j]$ has 2 definitions/uses and $Y[i',j']$, $\forall i',j' s.t. 0 \leq i',j' \leq N-1$ has one use. Hence $W(Z[i,j], Y[i',j']) = 3$ as shown in Figure 7 (a). The span or live range of $Y[i,j]$ is almost the entire matrix multiplication code (with $N-1$ uses). Hence, $W(Y[i,j], Y[i'j']) = 2N-2$. Note that we have not shown all the edges in the weighted interference graph in Figure 7 (a) to avoid the clutter.

Other Weight Measures: Note that not do we only care about the frequency of simultaneous accesses for the two variables (reflected in the sum of the number of accesses in weights in Figure 7), for a queue allocation having the same period of access is also important. A close approximation to the period of access would be the number of accesses. Hence for a fixed $s = a_{u\cap v}(u) + a_{u\cap v}(v)$, $W(u,v)$ should be higher for the case $a_{u\cap v}(u) = a_{u\cap v}(v)$ than for the skewed case where $a_{u\cap v}(u) = a_{u\cap v}(v) + c$. In $a_{u\cap v}(u) = a_{u\cap v}(v) = k$ case, all the uses and definitions of u (v) can be allocated to a queue $Q1$ ($Q2$) such that $n_1 = n_2$ (same pipeline depth). This will allow us to synchronize the accesses to u and v if they both happen to be operands to the same instructions. One way to do this is to multiply $a_{u\cap v}(u) + a_{u\cap v}(v)$ by the entropy of $\frac{a_{u\cap v}(u)}{a_{u\cap v}(u)+a_{u\cap v}(v)}$, $H(\frac{a_{u\cap v}(u)}{a_{u\cap v}(u)+a_{u\cap v}(v)})$ [Ham86]. ($H(x) = x\log_2(1/x) + (1-x)\log_2(1/(1-x))$). For example, given that u has 5 accesses and v has 5 accesses in the intersection of the two spans, $W(u,v) = (5+5)*H(1/2) = 10$. On the other hand if u has 2 accesses and v has 8 accesses, the old model still gives us the weight to be 10. However, $H(1/5) = H(4/5) = .72$. Another way to achieve the same effect would be to set $W(u,v) = a_{u\cap v}(u)*a_{u\cap v}(v)$. Note that $x*(1-x)$ maximizes at $x = (1-x) = .5$.

Another variant on weight computation is scaling (used in [CH84] for local allocation). We could divide the weights by the span of the intersection of the live ranges $u \cap v$ (in code size). This would reward the accesses that occur over a shorter time span.

Weighted Cliques: The next step is to identify cliques with large weights from the weighted interference graph. Let weight of a subset of vertices $V' \subseteq V$ be defined as: $weight(V') = \left(\sum_{u,v\in V'} W(u,v)/2\right)$. The division by 2 accounts for the weight of each edge being counted twice. Let *average edge weight* over $G = (V,E,W)$ be given by $w_{av} = (\sum_{(u,v)\in E} W(u,v))/(2*|E|)$.

Definition 4 *A set of vertices $C \subset V'$ forms a weighted clique if $weight(C) > w_{av} * (|C|^2 - |C|)/2$. $(|C|^2 - |C|)/2$ is an upper bound on the number of edges in C.*

The next step is to identify all the weighted cliques in the weighted interference graph $G = (V, E, W)$. Note that a set of vertices C could be a weighted clique by our definition, but may not be a clique in $G = (V, E)$ (if certain edges are missing, but are compensated by other edges of high weight). Finding all the cliques in a graph is also NP-complete [GJ79]. A greedy heuristic can be used to identify the weighted cliques (we have used this heuristic in [Tya96] and it performs well). The following procedure returns a maximal weighted clique C in G.

procedure Find-Weighted-Clique($G = (V, E, W)$, C)
  Let $(u, v) \in E$ be the edge with maximum weight $W(u, v)$. $C = \{u\}$; $s = 1$; $x = \{v\}$;
  s is the current size of the clique.
  while $(weight(C \cup x) > (((s+1)^2 - s)/2) * w_{av})$
    $C \leftarrow C \cup x$; $s \leftarrow s + 1$;
    $x \leftarrow \{w \in V \mid weight(C \cup \{w\}) \geq weight(C \cup \{u\}), \forall u \in V - C\}$.
    Note that only neighbors of vertices currently in C need be considered for w.
  end while
end procedure

Note that we need not find largest cliques in G since our queues are of fixed size. Let n_{max} be $\max_{Qi}[n_{Qi}]$, the depth of the longest queue. We may wish to limit ourselves to find cliques of sizes up to some small constant times n_{max}, perhaps $2 * n_{max}$. This would make clique identification more efficient.

SCHEDULING THE WEIGHTED CLIQUES ON QUEUES: The next step is to schedule the vertices in a weighted clique C on some subset of queues. The first step here is to build a sequencing graph for the nodes in a clique C. Figure 8 (a) shows a piece of a program with the corresponding live ranges. The corresponding clique in the interference graph is shown in Figure 8 (b). Note that we have multiplied the access counts for u and v to derive $W(u, v)$. The precedence or sequencing graph is also demonstrated in Figure 8 (b). It is a diamond shaped precedence graph due to reconvergent branches. A live range is given precedence based on the first access to its variable (hence $l1$ is the top node in the precedence graph). The precedence graph for a clique can be derived by performing a topological sort of its vertices. We need to find a cover of the precedence graph such that the precedence ordering is preserved and each cover element is allocated to one queue. The following heuristic steps perform the covering operation.

(1): Let $a_{\cup_{u \in C} u}(l_i)$ be the number of accesses to l_i value within the union of live ranges of all the nodes in the clique C. Figure 8 (b) shows these values: *e.g.* for $l1$ it is 3. Sort the live ranges by their access count and put them in separate lists in their topological order: List $L1$ gets $l1$ with access count 3 and List $L2$ gets $l2, l3, l4, l5$ all with access counts 2.

(2): Find an available memory queue with size closest to the number of entries in the list with highest access count, $L1$ in our example. Let us assume that 2 queues $Q1$ and $Q2$ with sizes 2 and 3 respectively are available. $Q1$ best matches the size of $L1$.

(3): Allocate this list to this queue, $L1$ on $Q1$. If size of the queue is bigger than the size of the list (true in this example), move nodes from the list with next highest access count with the best similarity with $L1$ nodes.

Definition 5 *The function* $similarity(u,v) = a_{u\cap v}(u) + a_{u\cap v}(v) + defs_{u\cap v}(u) + defs_{u\cap v}(v) + live_{u\cap v}(u) + live_{u\cap v}(v)$*, where* $defs_{l'}(l)$ *is the number of definitions of* l *reaching the live range* l' *and* $live_{l'}(l)$ *is the number of uses of* l *live at exit from* l'.

Similarity function indicates that the live range $l3$ is closest to $l1$ and hence queue $Q1$ is allocated $l1$, $l3$ or $x1$, $x3$ in that order.

(4): Allocate all the remaining unscheduled lists following steps (2) and (3). This results in $x2$, $x4$, $x5$ being allocated to $Q2$.

Figure 8 (c) shows the code resulting from this queue allocation. WR is a write and RD is a read. ADV denotes an advance operation. Note the advance operations needed in the alternate branch path in order to keep the queues synchronized. Another optimization pass can consolidate advances on the same queue so that the number of advances used is less than the queue size.

For the matrix-multiplication example, the precedence graph for the cliques derived for the 0th row of Z has the form (a linear chain) shown in Figure 9. For a high value of N (say 100), all these live ranges go to the same list $L1$ with access count N. Since it is unlikely that there exists a queue of size N, the queue of highest available size matches this list. Say $Q1$ of size 10 is available. Then the first ten entries (topologically) of $L1$ ($Z[0,0]$, $Z[0,1]$, ..., $Z[0,9]$) are allocated on $Q1$.

Hence the overall queue-allocation algorithm is as follows:

---

procedure Queue-Allocation(intermediate code I and flow analysis)
  Compute the weighted interference graph: $G = (V, E, W)$.
  Compute the Maximal cliques in $G = (V, E, W)$. Form the clique interference graph from G: $G' = (V_C, E', W)$.
  Color the clique interference graph G' where coloring a clique $C \in V_C$ corresponds to queue scheduling described earlier (potentially uses multiple colors) using a Chaitin style skeleton.
end procedure

Note that the most likely set of variables allocated to queues is the variables on the spill list of the register allocation phase (when queue allocation follows register allocation).

Integrated Register-Queue Allocation: Gupta *et al.* [GSS89] use clique separators in the interference graph to divide and conquer the register allocation problem. This method can be modified to provide an integrated register-queue allocator. The separator cliques can be allocated on the queues, while the separated halves could be register-queue allocated recursively. In this case, the separator clique vertices are not replicated in the two halves (as was the case in [GSS89]), which simplifies the allocation procedure. The criteria for separator clique selection would also be different in the integrated queue-register allocation.

2.5 Conclusions

Memory bandwidth is falling behind processing rates with advances in processor architecture and design techniques. One way to compensate for low memory bandwidth is to provide larger well-managed local storage. We propose to exploit available computing units in the processor to provide another form of local storage. This creates another trade-off, between larger on-chip memory (lower memory bandwidth) and computing rate. This trade-off can be undertaken dynamically under compiler control. An advantage of this trade-off is the ability to run the processor in *balanced* mode (or as close to it as possible) for a variety of programs due to dynamic reconfigurability of memory/computation units. The on-chip storage available cheaply is however not random-access. It can be used in FIFO (queue) configuration. This poses a new challenge for the compiler back-end. We propose several compiler techniques for allocating these queues, which fit in well with existing register allocation techniques.

References

[AS93] P. M. Athanas and H. F. Silverman. Processor reconfiguration through instruction-set metamorphosis. *Computer*, 26(3):11–18, March 1993.

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[BCKT89] Preston Briggs, Keith D. Cooper, Ken Kennedy, and Linda Torczon. Coloring heuristics for register allocation. In *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 275–284, Portland, Oregon, June 21–23, 1989. *SIGPLAN Notices,* 24(7), July 1989.

[BGG+89] David Bernstein, Dina Q. Goldin, Martin C. Golumbic, Hugo Krawczyk, Yishay Mansour, Itai Nahshon, and Ron Y. Pinter. Spill code minimization techniques for optimizing compilers. In *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 258–263, Portland, Oregon, June 21–23, 1989. *SIGPLAN Notices,* 24(7), July 1989.

[BGK95] Douglas C. Burger, James R. Goodman, and Alain Kägi. The declining effectiveness of dynamic caching for general-purpose microprocessors. Technical Report Technical Report 1261, University of Wisconsin-Madison Computer Sciences Department, January, 1995.

[BGK96] Doug Burger, James R. Goodman, and Alain Kägi. Memory bandwidth limitations of future microprocessors. In *Proceedings of 23rd International Symposium on Computer Architecture*, pages 78–89, May 1996.

[BTA93] J. Babb, R. Tessier, and A. Agarwal. Virtual wires: Overcoming pin limitations in FPGA-based logic emulators. In D. A. Buell and K. L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 142–151, Napa, CA, April 1993.

[CAC+81] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein. Register allocation via coloring. *Computer Languages*, 6(1):47–57, January 1981.

[CCK90] David Callahan, Steve Carr, and Ken Kennedy. Improving register allocation for subscripted variables. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 53–65, White Plains, New York, June 20–22, 1990. *SIGPLAN Notices,* 25(6), June 1990.

[CH84] F. Chow and J. Hennessy. Register allocation by priority-based coloring. In *Proceedings of ACM Symposium on Compiler Construction*, pages 222–232, 1984.

[Cha82] G. J. Chaitin. Register allocation and spilling via graph coloring. In *Proceedings of ACM Symposium on Compiler Construction*, pages 98–105, 1982.

[CM73] J. Cocke and R. E. Miller. Configurable Computer System. *IBM Technical Disclosure Bulletin*, 15:9:2847–2849, February 1973.

[Est60] G. Estrin. Organization of Computer Systems: The Fixed-Plus Variable Structure Computer. In *Proceedings of Western Joint Computer Conference*, pages 33–40, New York, NY, 1960. American Institute of Electrical Engineers.

[Gan87] D. Gannon. The influence of memory hierarchy on algorithm organization. In *The Characteristics of Parallel Algorithms*. MIT Press, 1987.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, San Francisco, 1979.

[GJG88] D. Gannon, W. Jalby, and K. Gallivan. Strategies for cache and local memory management by global program transformation. *Journal of Parallel and Distributed Computing*, 5:587–616, 1988.

[GSS89] Rajiv Gupta, Mary Lou Soffa, and Tim Steele. Register allocation via clique separators. In *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 264–274, Portland, Oregon, June 21–23, 1989. *SIGPLAN Notices,* 24(7), July 1989.

[Ham86] R. W. Hamming. *Coding and Information Theory.* Prentice Hall, second edition, 1986.

[HK81] J. Hong and H. T. Kung. I/O Complexity: The Red-Blue Pebble Game. In *Proceedings of ACM Symposium on Theory of Computing*, pages 326–333, 1981.

[HP96] J. L. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan-Kaufmann, 2nd edition, 1996.

[Kun79] H.T. Kung. Let's Design Algorithms for VLSI Systems. In *Proceedings of the Caltech Conference on Advanced Research in VLSI: Architecture, Design, Fabrication*, pages 65–90. Caltech, January 1979.

[Kun86] H.T. Kung. Memory Requirements for Balanced Computer Architectures. In *Proceedings of the 13th Annual Symposium on Computer Architecture*, pages 49–54. ACM, 1986.

[LRW91] Monica S. Lam, Edward E. Rothberg, and Michael E. Wolf. The cache performance and optimizations of blocked algorithms. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 63–74, Santa Clara, California, April 8–11, 1991. ACM SIGARCH, SIGPLAN, SIGOPS, and the IEEE Computer Society. *Computer Architecture News,* 19(2), April 1991; *Operating Systems Review,* 25, April 1991; *SIGPLAN Notices,* 26(4), April 1991.

[Mol83] D. I. Moldovan. On The Design of Algorithms for VLSI Systolic Arrays. *IEEE Transactions on Computers*, 71(1), January 1983.

[PF92] Todd A. Proebsting and Charles N. Fischer. Probabilistic register allocation. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 300–310, San Francisco, California, June 17–19, 1992. *SIGPLAN Notices,* 27(7), July 1992.

[RA78] T. G. Rauscher and A. K. Agrawala. Dynamic Problem-Oriented Redefinition of Computer Architecture via Microprogramming. *IEEE Trans. on Computers*, C-27(11):1006–1014, November 1978.

[RT91] J. H. Reif and A. Tyagi. An Optical Delay-Line Memory Model with Efficient Algorithms. In *Proceedings of the ADVANCED RESEARCH in VLSI : International Conference 1991.* MIT Press, 1991.

[Set75] R. Sethi. Complete register allocation problems. *SIAM Journal on Computing*, 4(3):226–248, 1975.

[Tya96] A. Tyagi. Integrated Area-Power Optimal State Assignment. In *Proceedings of the Synthesis and Simulation Meeting and International Interchange, SASIMI '96.* Daichisha Press, Kyoto, November 1996.

[WL91] Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 30–44, Toronto, Ontario, June 26–28, 1991. *SIGPLAN Notices,* 26(6), June 1991.