Segmenting The Dynamic Instruction Stream
For Distributed Computation

Benjamin Wallace
Yale College class of 2003
Department of Computer Science
Yale University
New Haven, CT 06520
benjamin.wallace@yale.edu

Abstract
As we reach the end of life for traditional microarchitectures [1] the search for computational performance has branched out in many different directions. The fields of automatic code generation for distributed computation, of producing multi-threaded hardware, and of embedding logic in memory all seek to increase performance. We propose a synthesis of these ideas, and explore the feasibility of running existing programs on it. We explore a method for program segmentation, motivated by an interest in the feasibility of running existing code on massively distributed systems with fast memory. We demonstrate that segmentation into a small number of segments is reasonably efficient.

1. Introduction
Our model for computing has remained essentially unchanged for the last 40 years: a Van Neumann machine with a memory, an arithmetic unit, and some control logic. Due to the rapid increase in transistor densities and physical bounds imposed by current semiconductor technology, memory access times and processor speeds have improved at very different rates. As a result a great deal of research has been put into caching, in an attempt to bring memory closer to the CPU, and thus decrease the cost of going to memory.

Recently there has been work trying to drive the CPU towards memory by embedding it in RAM. Current projects limit the processor to only a few megabytes of fast memory. Rather than having a single processor embedded in memory which accesses a larger backing store, why not have every portion of that backing store be addressed by a processor? If the control flow can pass between processors, we may achieve high bandwidth access to all portions of memory.

Certainly, this kind of architecture could be used as a cluster of parallel machines. Work has been done in automatic parallel code generation for message passing, grid computing, and spread memory architectures. This kind of compilation could likely produce binaries that would outperform standard sequential code for a given machine. However most of these methods require changes or restrictions to the source code of the program; from a transitional point of view it is important to ask how such an architecture might perform given today’s code base and compilation technology.

Could a modern program run on such a distributed architecture, and if so what would the cost be? In order to examine this question we construct the dynamic memory dependence graph. Several natural questions arise: What is the structure of the dynamic dependency graph? How is connectivity distributed over the graph? Are there natural divisions or low bandwidth interfaces that would enable a natural partitioning between sections of the graph?

In section 2, we will discuss related work in three fields and how it motivates this study. Section 3 lays out some background on the dynamic dependency graphs and describes the variation we used. Section 4 discusses building the graphs and presents data on the general structure of these graphs for our benchmark set. Section 5 formalizes the problem of breaking the program into sets, presents our method for doing so, and gives the resulting data. Section 6 discusses data duplication within the graph. Finally, Section 7 draws conclusions about the feasibility of this
kind of partitioning and discusses yet unanswered questions.

2. Related Work and Motivation

Increasing computational performance is an enormous field and we do not try to mention all of the recent trends here. Rather, we point to three specific trends that have motivated this study.

One direction of research is towards parallel computing and automatic parallel code generation. There have been numerous efforts to create programming languages, like Split C, and Titanium, which can easily be compiled into efficient parallel code [2, 3, 4, 5]. There have also been several attempts to generate parallel executables from code written in subsets of existing languages [6, 7, 8, 9]. These efforts have been remarkably successful when dealing with a subset of program types or when limited to a subset of the language in question. Both of these kinds of research address parallelism within a single task, trying to eliminate structural hazards by distributing the computation over multiple hosts. Though there are inevitable communication costs, these can often be masked by overlapping communication and computation.

Another trend in the commercial market is the trend towards multithreaded CPUs. Sun, Intel, Alpha, and IBM have all announced intentions to pursue this kind of parallelism [10]. Unlike the parallelism achieved by compilation for distributed computing, this trend addresses task-level parallelism. This kind of solution makes sense because it is difficult for a single thread of execution to fully utilize the resources of the processor, and because in modern computing environments, there are frequently multiple threads ready for execution. If history is any guide, the number of independent running threads will continue to climb as operating systems, programs, and interfaces become more complex.

The final and perhaps most important trend motivating our study is based on research done at the university of California at Berkeley on IRAM [11, 12, 13, 14, 15, 16]. The IRAM project grew from the observation that though CPU speeds where improving at 60% a year, DRAM access times where improving at only 7% a year. For some memory intensive applications a processor could spend 75% of its time waiting for memory [11]. This seems absurd in light of the 60% of their area and 90% of their transistor count that some modern processors devote to caching and otherwise optimizing access to memory. The IRAM project proposes unifying memory and computational resources. While there are fabrication challenges, such as improving transistor speed and adding more metal layers to the process, the potential gains are enormous. Latency to on chip memory decreases by a factor of 5-10, and bandwidth increases by a factor of 50-100 [12].

In practice, these kinds of chips have been shown to have a significant benefit in applications that are truly memory bandwidth limited. For tasks not limited by memory, their performance has been hindered by the slower clock speeds associated with memory fabrication [14, 15, 16].

If we combined these disparate ideas of fast memory access from embedding a processor on chip, the availability of multiple threads for execution, and an ever-improving ability to compile code for distributed execution, we end up with a CPU that is no longer centralized. Instead, we can imagine a grid of simple processors each of which has fast access to a small chunk of memory. This kind of computer could run many threads concurrently for task level parallelism, or run threads that utilize multiple nodes for instruction level parallelism. Certainly such a processor could run data-intensive and specialized media tasks extremely well. Sony corp. recently announced that it would collaborate with IBM and Toshiba to develop a cell or grid based microchip to power a future generation of videogame systems; though they have not released details on the project, then fundamental concept is the same.

The question we wish to answer, however, is not whether such a computer could run special-cased tasks quickly, but whether it could work for general computation, specifically with existing code. Thus, we must address the question of how much penalty is in-
curried when a program and its data do not fit on the memory addressed by a single processor. We do this by building the dependency graph for a set of test programs, and segmenting them into communication-efficient subsets.

3. Memory Dependence Graph

Our conception of a Memory Dependence Graph grew from work on Dynamic Dependency Graphs (DDG). Dynamic Dependency Graphs, at the microinstruction resolution, were introduced by Austin and Sohi [17]. Fundamentally, the DDG is a way to organize and group instructions from the dynamic instruction stream. Instructions are organized into an acyclic directed graph where each vertex of the graph represents a single instruction.

True dependencies are created any time there is a write to a location followed by a read of that data. These write-read pairs can arise between instructions operating on registers, or between pairs of store-load instructions coupled via memory.

Some of these dynamic dependencies are trivial. Modern processors try to alleviate write after write, or write after read dependencies. Some times these false dependencies cannot be eliminated due to insufficient hardware, such as a shortage of rename registers. Moreover, write after read dependencies through memory are not generally addressed at all. These standard cases are handled well by the DDG and do not constrain the graph structure. Besides these standard cases, there are other dependencies created by architected hardware limitations, which the graph cannot handle well. For example, if a function needs to utilize more values than fit concurrently in registers, the data will have to move in and out of main memory as needed. Thus, a large number of very simple dependencies would be created — all of which could have been avoided where there more registers available to the compiler. A similar situation arises for function arguments passed on the stack. Every argument exists somewhere in memory or registers before it is written to the stack and read back. Thus there are needless dependencies created for this copy to the stack.

In Austin and Sohi’s DDG, the graph was acyclic. This enabled them to generate information about the critical path length, and approximate graph width, while only tracking the leading edge of the graph (the most recently modification to registers and memory locations). Because we are interested in associating specific code sections to parts of memory and determining how interdependent they are we cannot just track the leading edge. Instead we track the entire graph. We build a Memory Dependence Graph (MDG) at an instruction, basic block, and function level. Each vertex in the graph represents word or a series of words in the instruction stream. Every vertex has a size, tracking how many words live in it. The directed edges are weighted to represent the number of words that one vertex requires from the other. Cyclic dependencies exist in the graph just as they do in the program. More formally:

\[ V \text{ is the vertex set, } E \text{ is the edge set} \]
\[ (a,b) \in E \text{ is an edge from vertex } a \text{ to } b \]
\[ w_{ab} \text{ is the weight of the edge from vertex } a \text{ to } b \]
\[ c_a \text{ is the size of vertex } a \]

4. Building the Graph

We built a simulator for the Alpha instruction set based on the Simple Scalar 3 toolkit [18]. The graph tools were written from scratch in C++. As the program executes, we construct the graph then perform analysis on it after it is complete. Graph generation consists of three major steps.

-Simulator

The simulator keeps track of the registers and memory touched in the current block of code (an instruction, a basic block, or a function). It then asks the graph to associate PCs to memory addresses in the case of loads, PCs to other PCs in the case of register operations that depend on other blocks and memory addresses to PCs in the case of stores. In order to track register dependencies, we keep a table of the most recent vertex to write the register.
-Dictionary
As the program runs, and asks for associations between addresses, the addresses are translated into unique graph vertex ID via a dictionary hash. The unique IDs are used to index into the vertex list. This translation process becomes especially useful since it enables simple address renaming and duplication of data and code segments.

-Graph
Because of the sparse nature of the graph, to conserve memory we keep an array of vertices, each with an edge list. As the program runs, the weights of edges accumulate as they are visited. We also keep track of loop back edges that connect a vertex to itself for the case of recursive procedures, or basic blocks that loop multiple times in sequence.

| Benchmark | Vertices | | | | |
|-----------|----------|----------|----------|----------|
|            | Total    | Inst per.| | | |
| bzip       | 1933461  | 215.58   | | | |
| chess-axp  | 1341218  | 186.30   | | | |
| compress-small-axp | 121707  | 1229.16  | | | |
| crafty     | 600671   | 3326.98  | | | |
| eon        | 143583   | 901.02   | | | |
| gap        | 8769621  | 46.02    | | | |
| gcc00      | 387648   | 562.55   | | | |
| gcc-axp    | 516575   | 351.81   | | | |
| go-2stone9-axp | 94464  | 1449.22  | | | |
| gzip       | 235772   | 653.80   | | | |
| ijpeg-axp  | 60944    | 31065.75 | | | |
| m88ksim-test-axp | 124005 | 4377.42  | | | |
| m88ksim-test-axp | 124005 | 4377.42  | | | |
| parse      | 1669693  | 318.27   | | | |
| perl-train-axp | 59218  | 768.70   | | | |
| pgp-axp    | 35002    | 5347.40  | | | |
| plot-axp   | 107047   | 2519.94  | | | |
| ss-axp     | 213175   | 562.48   | | | |
| tex-axp    | 208532   | 1139.16  | | | |
| vort00     | 2093977  | 387.26   | | | |
| vortex-small-axp | 1172308 | 214.99   | | | |
| vpr        | 387502   | 1464.16  | | | |
| wolf       | 28363    | 9852.14  | | | |

Table 1: highlighted benchmarks had high average connectivity

4.1 Instruction vs. Basic block vs. Procedure Level
We collected our initial data with all three levels of granularity. We tracked basic blocks in the dictionary via their exit address, and procedures via their entrance address. For procedures and basic blocks, we kept a list of loads and store and built associations when we left the block or function. To track dependencies through registers, we kept a table of the last block or procedure to write each register. For procedures, we also had to keep a stack of function calls to properly be able to handle returns.

We quickly found however, that basic blocks were preferable for analysis. Graphs at the instruction level became prohibitively large, and only a tiny portion of their vertices connected to more than one other vertex (more
will be said about this in 4.2). In addition, since the goal was to split the graph, it did not really make sense to break in the middle of a basic block. At the granularity of procedures, however, though the graph was considerably smaller, each vertex touched most of the registers, and many locations in memory. Again, since our goal was to eventually split the graph, it was not promising to have every vertex extremely connected. Thus, we primarily operated at the basic block granularity.

### 4.2 Basic Graph Data

We used a selection of benchmarks from the Spec95, Spec2000, and JBS benchmark suites. The wide selection was designed to capture as much diversity in the graph structures as possible. The programs were run with their standard input sets, and were run to completion unless otherwise noted. All testing was done on Yale’s Zoo computing cluster. Each node in the cluster had 1 gigabyte of main memory, which was necessary for building the graphs in memory.

A few benchmarks have unusually high average connectivity. As we will later see these present a challenge for partitioning. The distribution of this connectivity is also extremely interesting. For all the benchmarks, the vast majority of their vertices have few connections. It is interesting to consider, however, what percentage a graph’s edges come from vertices with each level of connectivity. For the most part, we observe three distinct trends, bottom heavy, top heavy, and split behavior.

![Figure 1: All benchmarks show a strong tendency towards sparsely connected vertices](image-url)
Figure 2: Percentage of edges from each class of vertex. We believe that these correspond to programs that utilize diverse data, and have no highly utilized loops.

Figure 3: Percentage of edges from each class of vertex. We believe that this class of benchmark has a frequented code section, or loop that dominates memory access. The data accessed also seems to be diverse.

Figure 4: Percentage of edges from each class of vertex. The majority of edges come from highly connected vertices. This indicates that the program loops a great deal accessing a limited subset of memory.
5. Partitioning the Graph

Next we would like to determine if there is some natural division of memory, which minimizes communication costs while staying well distributed. Let us begin by defining a partitioning $P$ and the communication cost.

a partitioning $P$ with $P=\{S_1,\ldots,S_n\}$ such that

for $i,j \in \{1,\ldots,n\}$ $i \neq j$, $S_i \cap S_j = \emptyset$

$S_1 \sqcup \ldots \sqcup S_n = V$

Specifically we would like to create a labeling for each vertex in the graph. The communication cost for that labeling is defined as:

\[
\text{communication cost} = \sum_{a,b \in E} w_{ab} \left( \sum_{i} c_i \right)
\]

The communication cost is trivially minimized when all of the vertices have the same label. Thus, we need to create a metric that takes into account the inequities in set size. We take the error in distribution to be:

\[
\text{error} = \sqrt{\prod_{i=1}^{n} \frac{c_i^2}{n}}
\]

To globally minimize this quantity over the space of all possible configurations is prohibitively complex. Complete exploration of the space, requires testing:

\[
\text{possible configuration count} = \frac{n^{|V|}}{n!}
\]

Given that we are working with graphs with millions of vertices, such a complete exploration is impossible. Instead, we develop a local heuristic to guide our search to an approximate solution. Given a vertex there are three pieces of information that inform the heuristic. First is the connectivity of this vertex to other sets e.g. how many dependencies do we have to each other possible set. Next, is the connectivity of each set to this vertex. We could have built a reverse graph by flipping all the edges.
to track which vertices depend on this one, but we found it was sufficient just to keep track of the weight of edges leading into each vertex from the same set and the weight leading in from other sets. This is computationally simpler and far more space efficient. We use this information to create a slight preference towards not changing sets. Finally, we note the current distribution of vertices in sets and use this information to ensure we have a reasonable distribution of vertices across sets. The weighing of these factors is somewhat arbitrary, and should be explored further in later research. A vertex being updated rates each possible set via:

When updating a $\square S_k$

let $r_a = \{x : (a,x) \in E\}$

and let $q_a = \{x : (x,a) \in E\}$

connection bias($i$) =

\[
\frac{\sum_{b \in S_k} w_{ab} + \sum_{b \in S_k} |r_a \cap S_k|}{|r_a|}
\]

size bias($i$) = $\frac{n - \frac{|C|}{n} \sum_{b \in S_k} |r_a \cap S_k|}{n}$

ing rating($i$) = connection bias($i$) + size bias($i$)

The coefficients of $\square$ and $\square$ were chosen as 0.6 and 1.0 respectively. The base algorithm randomly assigns a set to each vertex and then loops over the vertices updating them via our heuristic. After looping over the vertices, we evaluate our performance metric, and track the best configuration. We found that this base algorithm quickly falls into local minima and developed several simple improvements that improve performance dramatically.

**Randomize the update**

Rather than looping over the vertices in order, we select one at random to update. This mediates the problem of falling into cyclic updates that never converge. This is similar to approaches used to improve convergence in Neural Networks.

**Re-randomize if stable**

If the set configuration has reached stability (less than 1-5% percent of the vertices changed sets on the last iteration) then we randomly reassign the sets of a random set of vertices.

![Convergence Behavior](image)

**Figure 6**: We randomize the labeling for a test graph, then run the update algorithm for between 1 and 1000 iterations.
Simulated annealing
To help convergence over time and prevent us from moving too far from a global minima if we approach one, the number of vertices re-randomized is inversely proportional to the remaining number of updates to be performed.

Partial gradient decent
When we re-randomize, we also let a random set of elements be reassigned to the optimal condition seen thus far.

Figure 6 shows the relative performance of various improvements. Because the partitioning is completely re-randomized before generating each point, we do not see clean convergence. Rather, we see an increasing probability that the partition will converge to a better metric value with the additions to the base method.

5.2 Partitioning Data

We collected partitioning data by running our portioning algorithm on the graphs in memory. We ran it for an increasing numbers of sets from 2 to 128. In all cases, we used 2000 iterations of the algorithm. Each iteration updates one random vertex for every vertex in the graph. Because we are constructing an approximation, the data will always be an upper bound on the communication needed. It is possible that we could construct partitions that would yield lower total communication costs.

Figure 7 shows us a couple interesting things. First, at the low end of the graph, we see that we can construct a partitioning for two sets with next to no communication cost. Some benchmarks suffering less than 1% communication penalty per instruction. The communication cost in general seems to grow with the log of the set size. This seems promising for our ability to break up program execution into up into a small number of sets. We will discuss this more in section 7.

Because of the random nature of the update algorithm, no two runs of the data will end up with the same data. To verify that this random nature did not affect the results too much, we ran the same benchmarks multiple time

<table>
<thead>
<tr>
<th>Percentage of Instructions Needing Communication</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Graph showing percentage of instructions needing communication" /></td>
</tr>
</tbody>
</table>

Figure 7: The average appears logarithmic with set count
2 1.30% 532.89
4 1.97% 107.53
8 3.33% 60.76
16 2.88% 61.27
32 3.59% 53.59
64 4.08% 59.07
128 4.96% 44.90

Figure 8: The variance between communication costs computed by each run. We also note the average number of instructions executed before we switch from one set to another.

Overall, the variation seems acceptable. This variation could be reduced by running the partitioning algorithm for a larger iteration count.

The variability tends to increase with the number of sets, which makes sense since the complexity of the search space grows as well.

Figure 9 shows the absolute values of the error function for each of the set sizes. The error tends to decline as the set count grows because our heuristic weights the disparity between set sizes based on their percentage deviance from the optimal set size. Thus with more sets the optimal is smaller, and the weight for a given absolute deviation is larger.

It is somewhat difficult to graph the distribution of code and data in each of the partitioning of each of the benchmarks. Thus, I will just present some summary numbers, and a few examples.

![Error in Set Sizes](image)

Figure 9: Absolute error for each set size

![Code size vs data size in each set](image)

Figure 10: Shows the distribution of code and data for partitions into 2 sets
To give an idea of the relative code and data counts, when dividing the program into two sets, on average 81.7% of each set was data. Code tended to be distributed rather unequally. For two sets, the set with more code, contained on average 82.3% of the code, for four sets the number drops to 63.7% of the code, and for 8 sets it drops to 49% of the code. Still, the distribution remains highly skewed: for 8 sets, on average the sets were 11.7% off from an even code spread.

Figure 10 and 11 are typical of all the distributions, though as previously noted, the total sets sizes tend towards equality as the number of sets grows. The 80% - 20% distribution of code in interesting in that it tends to reinforce the adage that 90% of the time is spent in 10% of the code.

6 Data Duplication in the Graph

There are some simple cases where the ability to duplicate data or code, would simplify the construction a split graph and potentially improve the communication costs. It is easiest to look at an example of where this can help.

Consider the case in Fig. 12a. Due to the high communication cost from B to A and C to A, we would be forced to C and B in the same set. But if we split A into A and A’ as in Fig. 12b, B and C can live in different sets without problem. This gives us greater freedom in constructing the sets. There is a simple way to identify vertices that should be split in this manner:

if \( A \) and \( B \) \( \not\in V \) such that
\[
W_{AC} \geq W_{AB} > \bigcup_{x \in r_x} W_{cx}
\]

then \( C \) should be split.
Algorithmically we achieve the splitting by iterating over the vertices and storing the sum of their outgoing edges. For each vertex, we then check the weights to each of its outbound edges against the stored total for that edges destination vertex. If the weight of the edge exceeds the sum for the edge’s destination, we mark the vertex. If the vertex was already marked, we split it.

Splitting greatly reduces the average connectivity, but does not impact the communication cost as much as we originally expected it would. On average, the average connectivity falls by 22%, but the communication cost falls by only 1.6%. The only two benchmarks that markedly benefit from splitting the graph are gzip (by 7.2%) and parse (by 5.9%). These two both duplicate a fair amount of data and code when split. The gain from splitting vertices seems to affect the communication cost more as the number of sets increases, but still, the improvement is not much in absolute standards.
Partitioning also does not affect code size much. On average, the code size is nearly static, only growing on average 8.3% in the split graphs. The only outlier data seems to be Crafty, which duplicates 35% of its code. Data duplication is far more frequent, with on average 55% of the data being duplicated.

7. Conclusions

Based on the data we have seen, it seems likely that a program could be segmented into several sets, particularly if it need only be divided 2 to 16 ways. It seems that data and code duplication are not particularly important to finding good partitions, and are likely not worth worrying about.

We can compare this to traditional cached architectures by considering the following scenario. Taking conservative bounds on the performance of memory operations, let's say that access to on-chip memory is 5 times lower latency than traditional access to main memory, and communication between chips is as slow as access to traditional main memory. Access to on-chip memory will not be as fast as access to an on-chip L2 cache in traditional architectures. Say that the cache is 2 times as fast as on-chip memory. Say both chips have identical small L1 data and instruction caches, and assume that they run at the same effective core speed (in order to isolate relative memory speeds we must assume this).

<table>
<thead>
<tr>
<th>Memory Type</th>
<th>Relative Delay</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trad. Main mem.</td>
<td>100</td>
</tr>
<tr>
<td>IRAM off chip</td>
<td>100</td>
</tr>
<tr>
<td>IRAM on chip</td>
<td>20</td>
</tr>
<tr>
<td>Trad. Cache</td>
<td>10</td>
</tr>
</tbody>
</table>

Figure 16: Summary of hypothetical relative memory access times

Consider a program split over 8 processors with average (9%) communication cost. To continue to be conservative, we will assume that no communication can be overlapped with data transmission, though this is almost certainly not the case. Then the traditional micro-architecture would have to hit in the L2 cache at least 83% of the time to be as fast. If the traditional L2 cache is 5 times as fast as on chip IRAM, the traditional CPU need only hit 78% of the time to be as fast.

These cache-hit rates are certainly achievable for normal programs. In fact, given these conservative estimates, a program hitting in the L2 cache 95% of the time would take roughly half of the time to run on a traditional machine. Still, given the potential increase in parallelism available, this kind of cost may be acceptable.
7.1 Questions Yet Unanswered

Several questions remain unanswered. First, the choice of weights for the heuristic is hardly scientific. Further exploration into the effect of these weights on the convergence process is necessary. Perhaps improved heuristics could further reduce the communication costs, associated with partitioning.

The next big question is: can you predict how to partition a program at compile time, at load time, or even while running. It seems likely that you could predict the communication costs of calling a function with reasonable accuracy, given some profiling information. Thus, it should also be possible to create balanced sets at compile time.

Special thanks to Dan Friendly of Yale University’s Computer Science department for his help and guidance throughout this project. And to Arvind Krishnamurthy for his advice on graph algorithms.

References


