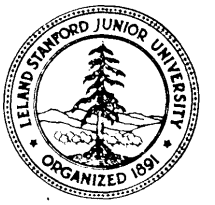
COMPUTER SYSTEMS LABORATORY

STANFORD UNIVERSITY · STANFORD, CA 94305-2192




# An Overview of the MIPS-X-MP Project

**John L. Hennessy and Mark A. Horowitz**

**Technical Report No. 86-300**

**April 1986**

The MIPS-X-MP project has been primarily supported by the Defense Advanced Research Projects Agency under contract # MDA 903-83-C-0335. The National Science Foundation has supported work on the project under Presidential Young Investigator awards to the authors. Several of the key project members are supported by a variety of other sources including: Natural Sciences and Engineering Research Council pre- and post- doctoral awards from Canada and Digital Equipment Corporation external research grants. Additionally, the SUNDEC project, funded by Digital Equipment Corporation, has provided many of the computers that were used to carry out this work.

# An Overview of the MIPS-X-MP Project

**John L. Hennessy and Mark A. Horowitz**

**Technical Report No. 86-300**

**April 1986**

**Computer Systems Laboratory**
**Departments of Electrical Engineering and Computer Science**
**Stanford University**
**Stanford, California 94305**

## Abstract

MIPS-X-MP is a research project whose end goal is to build a small (workstation-sized) multiprocessor with a total throughput of 100-200 mips. The architectural approach uses a small number (tens) of high performance RISC-based microprocessors (10-20 mips each). The multiprocessor architecture uses software-controlled cache coherency to allow cooperation among processors without sacrificing performance of the processors. Software technology for automatically decomposing problems to allow the **entire** machine to **be** concentrated on a single problem is a key component of the research. This report surveys the four key components of the project: high performance VLSI processor architecture and design, multiprocessor architectural studies, multiprocessor programming systems, and optimizing compiler technology.

**Key Words and Phrases:** processor architecture, RISC, multiprocessors, compiler systems, single-assignment languages, LISP

# Table of Contents

1 Introduction 1
2 MIPS-X: a High Performance VLSI Processor 2
2.1 Second generation RISC machines: challenges 2
2.2 The MIPS-X approach 5
3 Multiprocessor Architecture 6
3.1 Issues 7
3.2 The Memory Hierarchy 11
4 Multiprocessor Programming Systems 12
4.1 Parallel programming languages 12
4.2 Automatic partitioning 13
4.3 Optimization of expression-oriented programs 16
5 Advanced Compiler Technology 16
5.1 Problems and challenges 17
5.2 LISP on a RISC (?) 17
5.3 Profile-based compilation 19
6 Conclusions 22

# List of Figures

**Figure 1: Possibilities for Address Translation** 4
**Figure 2: A Shared Memory Multiprocessor** 8
**Figure 3: Throughput versus Processor Count** 9
**Figure 4: Throughput versus Processor Count (Pipelined Bus)** 10
**Figure 5: The SAL Compilation System** 14
**Figure 6: Speed on LISP Benchmarks (Preliminary)** 18
**Figure 7: The MIPS UCODE Compiler** 20

## 1 Introduction

This paper surveys the MIPS-X-MP project The theme of the MIPS-X-MP project is high performance, low cost computing through a small[1] multiprocessor. This machine should achieve throughput in the range of 100-200 mips using tens of processors. A key goal is to provide this throughput level on single problems with adequate parallelism. The machine should also be suitable for clustering to allow more processors to be used, when the restricted communication between clusters does not pose a serious performance limitation.

The research goals of this project are to explore:

- The design of a multiprocessor using substantially higher performance processors than other machines. Designing a memory hierarchy that allows consistent sharing while supporting tens of processors of this performance level is a key issue.
- The architecture and implementation of a second generation RISC microprocessor. While our initial goal is to design a machine in the 10-20 mips sustained range (1 mip = a VAX 1 1/780), we would like the architecture to grow gracefully to 40-50 mips.
- Software systems for programming multiprocessors; we believe that this problem is as challenging as the design of a multiprocessor and has received less attention.
- The development of next generation compiler technology both for conventional procedural languages, as well as LISP and purely applicative languages.

These four research goals define the major aspects of the project. Each area is reviewed in this paper. However, this document is not meant to discuss or define all of the aspects of the project. Instead, it serves to give an overview and provide citations and context for a number of project components that are key to achieving these goals.

## 2 MIPS-X: a High Performance VLSI Processor

The first generation of RISC machines (the IBM 801, the Stanford MIPS, and the Berkeley RISC) explored the basic principles of streamlined architectures. The Berkeley and Stanford projects produced machines capable of performance in the range of one to two times a VAX 1 1/780 on nonfloating point benchmarks. They concentrated on languages such as C and Pascal and attempted to develop both software and hardware technology needed to make high performance VLSI processors.

---

[1] 'Small refers to the number of processors and the size of each processor (i.e. they are VLSI processors), and hence, the size of the overall machine.

## 2.1 Second generation RISC machines: challenges

The key challenges for the second generation of VLSI RISC engines are:

- To provide substantial performance improvement by more aggressive pipelining and higher instruction execution rates.
- To address floating point computation. A high performance coprocessor interface is used to achieve this in MIPS-X and we will briefly discuss the approach in the next section.
- To explore the architectural needs, if any, of other language environments, such as LISP. We discuss our LISP experiments in the section on compiler activity.
- To explore the impact of a multiprocessor configuration on the uniprocessor architecture and organization. We discuss many of these issues in the next section.
- To develop the next generation of software technology needed for such machines. As we will see, achieving higher performance requires more reliance on hardware-software integration; we discuss these requirements in this section and address our approach in the section on compiler technology.

To understand the issues involved in maximizing the CPU compute performance, let's examine a simple performance equation:

Performance = (clock speed) / (instruction count * cycles per instruction)

RISC machines attempt to maximize performance by producing improvements in clock speed (factors of 2-5, typically) and major improvements in the cycles per instruction (factors of 5 to 10). They allow a slight increase in the instruction count (less than a factor of 2). A key issue is to examine how the instruction count is impacted for languages that may have different characteristics from the languages that have been studied to date: for example, LISP and other applicative languages may be suitable for special instructions or architectural support that can significantly affect the instruction count without significantly increasing the clock cycle or the cycles per instruction. However, for procedural languages the instruction count tends to vary little among architectures.

Ideally, a RISC machine attempts to drive the cycles per instruction measure to 1. We are concerned with the *throughput rate* on instructions; the machines are pipelined and instructions actually take several cycles to complete. Single-cycle execution means that instructions are initiated and completed at the rate of one per cycle. This ideal goal can be fairly easily met for register-register instructions (about 40-50% of the instruction mix). The hard cases are branches and memory access insauctions. For example, the Clipper microprocessor does register-register ALU operations in a singlecycle, but takes 4 to 10 cycles for loads/stores and branches. This makes the average number of cycles per instruction in the range of 3 to 4 (the same holds for the IBM PC-RT). MIPS-X comes very close to achieving the single cycle goal, if we ignore external degradation. External degradation is primarily caused by cache misses, but TLB misses and other stalls also contribute. On the VAX 1 1/780 such stalls increase the cycles per instruction ratio by roughly 25% (i.e. two cycles per instruction). On a RISC machine achieving single-cycle execution, a degradation of one cycle per instruction, halves the performance of the machine. Hence the criticality of

a high performance memory subsystem.' What are the major factors influencing the clock speed? The two most significant factors are memory access and critical control paths. By reducing control complexity, a RISC machine minimizes most of the key internal critical paths: the single cycle discipline coupled with state update only once at the instruction's completion, simplify the pipeline and interrupt control. Clock speed is limited by the necessary cache access time; in effect, **a** well-designed architecture should be memory limited. Both the bandwidth and latency of memory should pose performance limitations. Thus, to maximize clock speed we must make addresses available as **soon** as possible and minimize the overhead in accessing the cache.

The interaction of **address** translation and cache access affects the access time to memory. Figure 1 shows how one can **choose** to do memory mapping. If the address translation is before the **cache,** then it slows down the cache access, impacting the machine cycle time. **A** common technique is to put the translation in parallel with the cache, using only the unmapped page offset bits to index the cache and doing hit detection with the translated bits when they come out of the TLB (Translation Lookaside Buffer). The main disadvantage of this scheme is that the number of lines (or sets) in the cache is limited by the page size, because only the unmapped bits can be used to do the cache look-up. The **cache** size can be increased by using more associativity, but the overhead cost of additional degrees of associativity is **nontrivial.** The final scheme uses **caches** on the virtual, rather than the physical address. Virtual data caches can be difficult to manage, because of the synonym problem that occurs when a shared data word is mapped to two different virtual addresses, resulting in two different copies. It is possible to avoid this problem in software, but it may be difficult for existing software systems. We will discuss the impact of these schemes in a multiprocessor context in the next section.

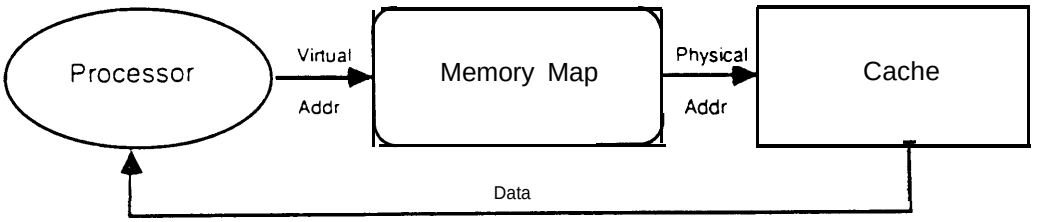
Many people **outside** of the RISC arena have stated that RISC and floating point are incompatible. The basis for this claim lies in the fact that floating point is inherently a multicycle operation. Of course, most RISC machines implement their instructions in multiple cycles; the basis of the single cycle approach is single cycle *instruction initiation.* The RISC ideas can easily be extended to floating point using a multicycle floating point coprocessor that initiates operations in a single cycle.

---

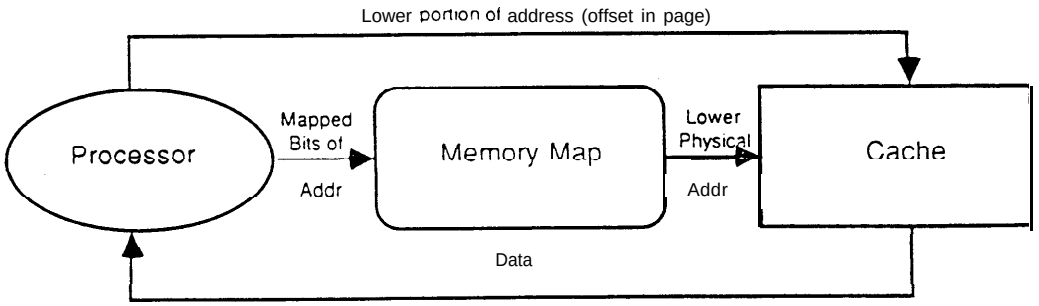
[2] Actually, this degradation increases relative to performance; if a stall (say a cache miss) takes a fixed amount of time, then the faster the machine, the more significant the effect

Figure 1: Possibilities for Address Translation

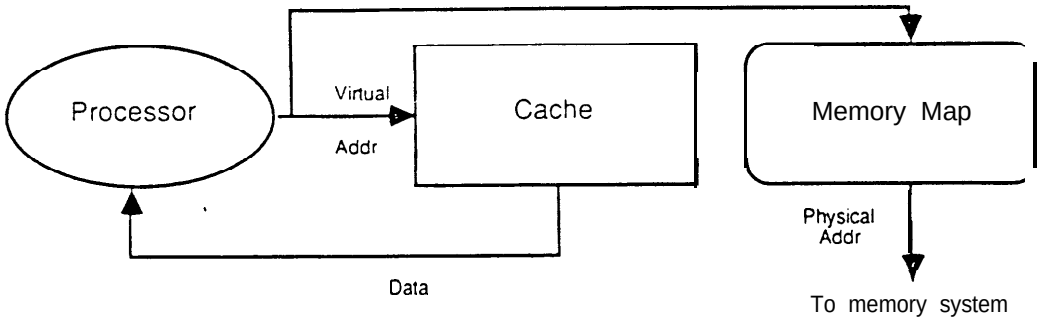
Scheme 1: Map before Cache




Scheme 2: Map in Parallel with Cache




Scheme 3: Virtual Cache



## 2.2 The MIPS-X approach

We now discuss how the MIPS-X architecture addresses these issues. A key challenge in the MIPS-X design is to achieve single cycle execution rates for load, stores, and branches, which all together account for 50% to 70% of the instruction mix. MIPS-X is in many ways simpler than the first MIPS processor. In particular, the instruction repertoire and formats are both much simpler, this is key to achieving higher performance. The memory subsystem and interface are more sophisticated.

To reduce the instruction bandwidth requirement, which accounts for about 70% of the memory bandwidth, MIPS-X uses an on-chip instruction cache. By using a large block size and a smaller sub-block size, MIPS-X can have a 512 instruction cache when fabricated in a 2 micron CMOS technology. High associativity (8 sets or buffers) allows high hit rates in the range of 80% to 90%. In comparison, the 68020 fits a cache of one-tenth the size and has a larger die[3]. In MIPS-X a miss on the internal instruction cache causes a fetch back of two words from the external cache (combined instruction and data).

Because of the speed of the on-chip instruction cache, MIPS-X assumes that off-chip data access will take longer (1.5 cycles versus about 1). To maximize the time available for the external cache, MIPS-X uses a single addressing mode: base register plus offset. This simple addressing mode allows the computation of the effective address to begin very early; a special address adder is used to minimize the time needed to compute the effective address, thus maximizing the cache access time.

The other major challenge to maintaining single cycle execution is the cost of branches. We have investigated and compared a variety of branching schemes [7]. MIPS-X uses an extension of the original delayed branch scheme, introduced in all three of the first RISC machines (the IBM 801, MIPS, and Berkeley's RISC). The MIPS-X scheme is called "squashed branches". In essence, it requires compile-time branch prediction to choose whether a branch will be taken or not. The branch delay slots are then scheduled appropriately. In MIPS, we found that scheduling optimized code became very difficult, because of the absence of harmless operations[4]. Due to its deeper pipeline the natural branch delay for the MIPS-X compare and branch instruction is 2 delay slots. The squash bit, included in the conditional branch instructions, indicates that a miss-predicted branch should cause the instructions in the branch delay slot to be squashed, i.e. no state update should occur from those instructions. This simplifies the task of finding candidates to fill the branch delay slots. As shown in [7], the squashed branch scheme together with

[3] The higher instruction density and more powerful instructions of the 68020 improves the effectiveness of the cache over MIPS-X. Although this factor is not known exactly, it is probably between 1.7 and 25, giving MIPS-X a cache effectively 4 to 6 times larger.

[4] The branch delay slots can only contain instructions that will not affect the computation when the branch is mispredicted. Good optimization and register allocation eliminate many redundant computations and tighten the. register allocation, reducing the frequency of such instructions.

software profiling performs comparable to many far more complex hardware schemes.

MIPS-X uses a virtual external cache so as to minimize the overhead in accessing that cache. A large external cache (64K to 256K) is planned, thus the cache miss frequency should be low, and address translation overhead should be incurred only rarely. The importance of good cache performance for a high performance RISC machine cannot be overestimated[5]. We have a major investigation of cache performance ongoing. Modeling caches of this size is challenging, since many small benchmarks nearly fit in the caches. Operating system interaction is also important and standard cache modeling techniques have ignored **or "guessed** at" the effects of the operating system. Thus, the cache measurement and modeling activities are aimed at obtaining very large traces with multiple user processes and operating system **activity;** the initial work on a technique to collect such traces is described in [2].

MIPS-X uses a RISC-like floating point coprocessor interface.

## 3 Multiprocessor Architecture

In this section we discuss our motivation for a shared memory machine, then we discuss the key design problems, and finally the approach that we are using for MIPS-X-MP. Our motivation for a shared memory machine built of small numbers of fast processors is based on several observations:

1. Many problems contain only limited amounts of parallelism that can be efficiently exploited For example, recent studies have shown that some problems thought to be highly parallel in fact had rather limited parallelism [6].
2. Even in applications where reasonable amounts of parallelism are available, significant serial, or nearly serial, code segments may occur. A problem where 90% of the program can obtain a speed-up of 10, and the remaining 10% **has no** parallelism, will actually run only about 5 times faster.
3. The effectiveness of exploiting parallelism is inversely proportional to the cost of communication; that is, lower granularity parallelism requires more communication per unit of computation. A shared memory machine is most effective at providing high bandwidth communication among a small number of processing units.
4. **A** shared memory model is attractive for its simplicity. Such a machine can easily be used with a message passing paradigm, providing efficient buffering and communication between processes. Alternatively, sharing of large data structures in a common address space is also possible.

The key disadvantage of a shared memory multiprocessor is its lack of expandability. Expandability depends on the bus bandwidth of the shared bus, the utilization of the bus by the processors, and the design tradeoff between bus expandability and bus performance. The latter problem places practical limits on buses connecting high performance machines. However, this limit on expandability can be overcome by a second level interconnect media that connects clusters on a shared bus. Of course, this change in interconnection strategy introduces a dichotomy into the architecture both in communication bandwidth as well as communication protocol. Due to the effect of locality of

[5] There is also a significant impact on a shared memory multiprocessor, as we will see.

communication that is present even in nonshared memory designs, this problem of differing communication bandwidths and costs must be faced in any expandable machine. Because the view of shared memory is that it is a controlled resource for sharing information, the difference in communication on the bus and between clusters should not appear to the programmer. Whether software can deal with this partitioning effectively is an interesting open question.

## 3.1 Issues

In Figure 2, we show a typical design for a shared memory multiprocessor. There are a number of challenging design issues in such a machine. These include: cache coherency, address translation, a secondary communication bus, and process synchronization.

Most shared memory machines being built both commercially and in research projects use hardware cache coherency (or a shared cache as in the Alliant). These approaches are attractive because they impose no burden on the software. They work fine with the relatively slow processors being used; however, in **a** high performance RISC machine, the access path to the cache is the critical timing path in the machine. The cache coherency hardware slows down that path. In addition, the atomic cache coherency offered by these hardware schemes provides a more tightly coupled structure than necessary. In most cases, the processors will need some higher level of synchronization, at those less frequent synchronization points that data may need to be interchanged or access to a shared data structure may need to be synchronized. Supplying caches **that** guarantee synchronization at a finer level will not improve the system performance in these cases. In fact, **a** tighter model of coherency may increase bus traffic, which can have negative effects.

For a high performance shared memory multiprocessor, bus and memory bandwidth limitations pose the most significant performance limitations. Figures 3 and 4 show the effects **of bus** contention, by plotting effective performance versus processor count[6]. We assume that a memory cycle over the bus takes ten processor cycles (a relatively ambitious assumption). Curves are shown for a variety of reference rates; a 1% reference rate means that 1% of the processor's cycles generate a bus transaction. This reference rate must include cache misses and any cache coherency traffic that generates a bus cycle. Figure 3 shows the data for a nonpipelined bus, while Figure 4 shows the data for a bus with the same latency but pipelined two stages deep. The throughput of such a pipelined bus is twice as much, but the latency may be worse (assume we assumed the same in the plots), because of the overhead of pipelining.

[6] These performance models and calculations were done by Malcolm Wing.

Figure 2: Shared Memory Multiprocessor

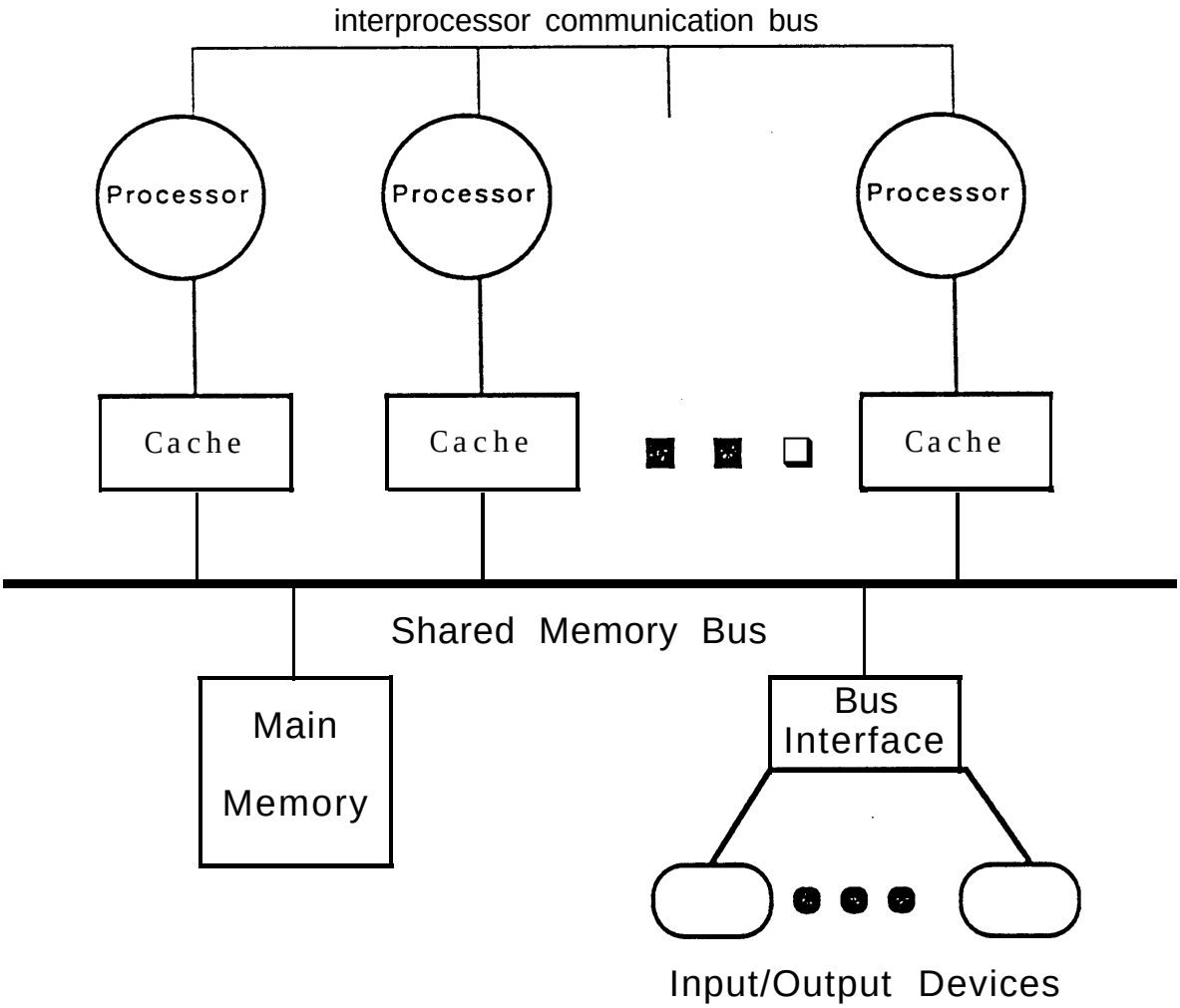


Figure 3: Throughput versus processor count

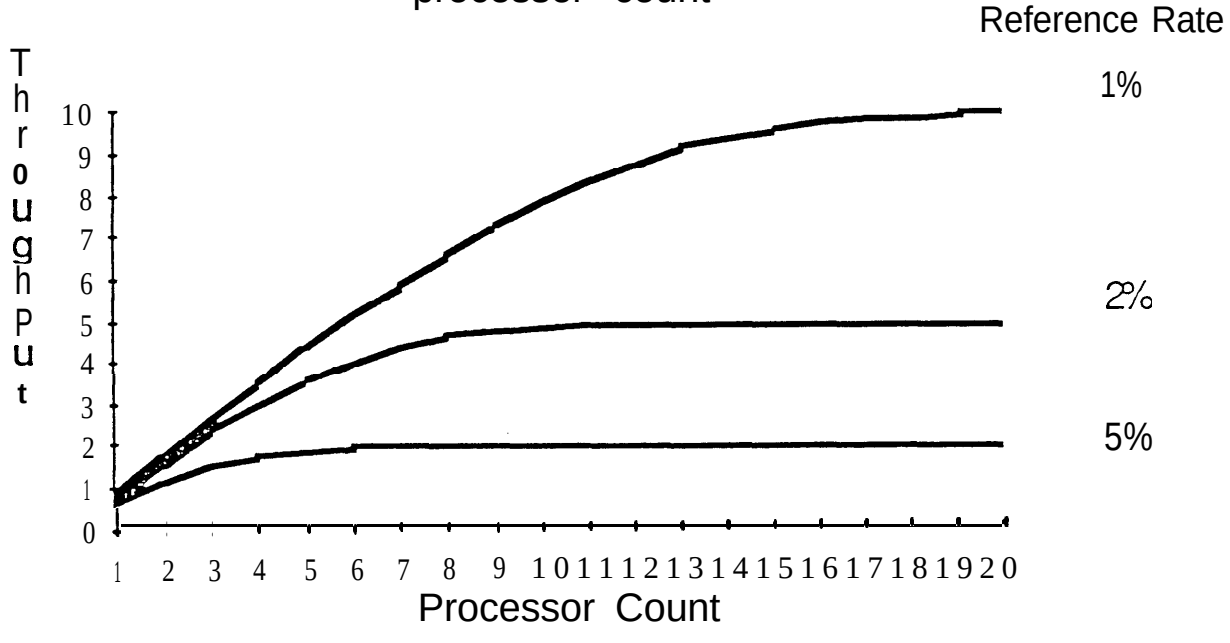
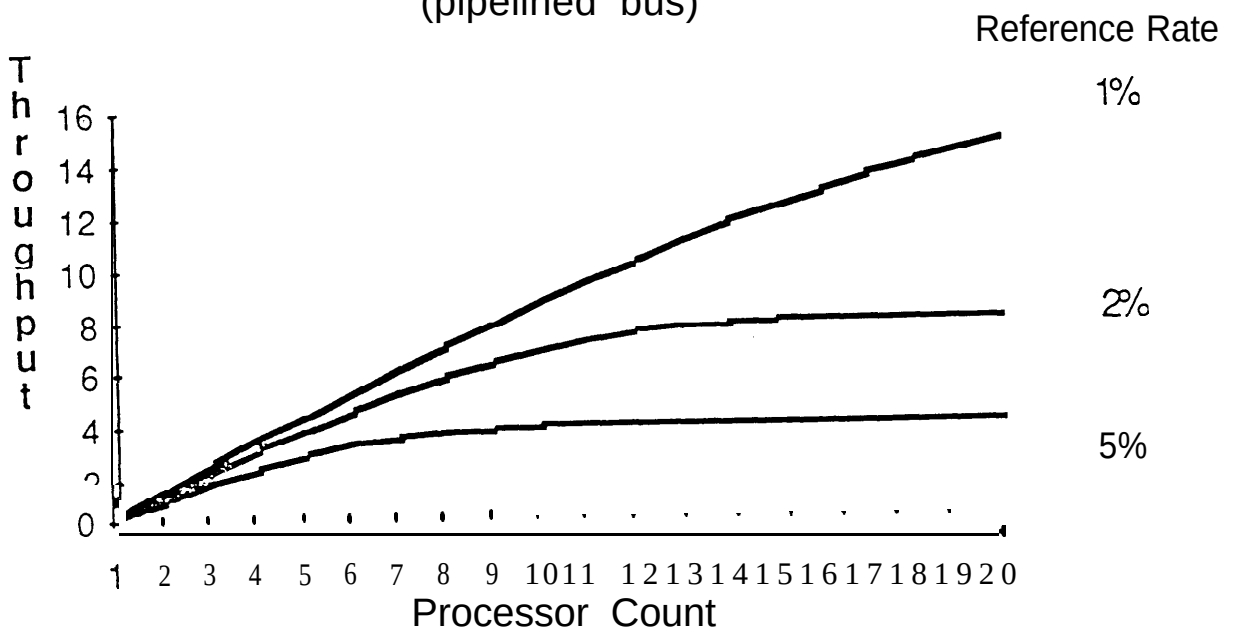


Figure 4: Throughput versus processor count (pipelined bus)



### 3.2 The Memory Hierarchy

The key architectural issue for this style of multiprocessor is the design of a high performance memory hierarchy and access bus. To minimize the cache access overhead., we have chosen to use a virtual cache. Once one chooses a virtual cache there are two other key issues that need to be addressed: what cache coherency hardware, if any, is appropriate, and how to do memory mapping.

MIPS-X-MP has chosen to use software-controlled caches rather than a hardware cache coherency mechanism. This choice avoids two potential performance problems. First, by not implementing cache coherency we can avoid dual porting the cache. This dual porting has a significant impact, especially if the main bus is not synchronous with the processor. Second., because cache coherency offers atomic consistency, it may generate more bus traffic than a mechanism that requires less tightly coupled caches'. Of course, we must ensure that our software-controlled mechanisms will work well in this environment. This means that we must know what portions of memory are shared and when a consistent view of shared memory is required.

To make the software enforced coherency efficient we need some hardware support over cache control. We need the ability to invalidate a portion of the virtual address cache, to force a new copy to be fetched from memory. Also, since the caches need to write-back', we also need a mechanism to **force** some portion of the cache to be written back to main memory. These invalidate and write-back operations can be tied to the synchronization points among the processes. Furthermore, since these operations can be done synchronous with the processor, they need not interfere with processor's attempts to access the cache[9]. Alternatively, we can mark a page as noncacheable; this will eliminate the cache coherency problem The noncacheable approach is efficient when data is read or written only once by a process between changes or use by another process. The Nebula design [3] also uses software controlled caches, but uses a very large block size and treats a block like a page.

The question of how to translate addresses remains. In a multiprocessor, the issues of where to place the TLB (the memory translation hardware) is tricky. If the caches are virtual it can be placed after the caches; however, all the TLBs must be kept coherent, so that a consistent view of memory is kept. Some proposed multiprocessors handle this problem with special hardware. The SPUR [14] design at Berkeley uses another approach: a virtual TLB [15]. The memory mapping tables are kept in virtual space and every cache miss requires a translation using these tables. Since cache misses will usually be much more frequent than translation misses in a hardware TLB (a

---

'This really depends on what cache coherency mechanism is used; the more complex mechanisms generate less traffic than simpler schemes.

[8] The write traffic alone from three MIPS-X processors would swamp a typical high-performance bus.

[9] By watching the external cache bus, we can determine when the processor will access the cache and avoid interference.

factor of 10 is a typical difference), the cost of the translation must be **kept low, otherwise this software scheme will** perform very poorly compared to hardware schemes. Keeping the translation miss penalty low **will** probably require special-purpose hardware to handle cache misses and to translate the missed virtual address to a **physical address. If** the system implements an atomic cache coherency model, then a beneficial symbiosis occurs, because the coherency mechanism **will** keep **all** the copies of the cached mapping tables up-to-date.

MIPS-X-MP uses a different approach: the TLB is placed at the main memory. The **TLB** can handle **all** page mapping as **well.** Only one copy of the map tables exists and that TLB processor can also track the use of shared memory, because **all** requests for access must come through the processor. Because only a single TLB exists, we can afford to make it a high performance, pipelined map and memory interface. Additionally, the TLB can track access **writes** and **cacheability.** It could **also** keep a directory of the locations of shared pages. **All** of these schemes will lower the overhead of maintaining a consistent view of shared memory.

## 4 Multiprocessor Programming Systems

Despite **all** the activity on multiprocessor architecture, too little research has been done on **general-purpose, parallel programming** systems. This is ironic, since good architectural design relies on the **availability** of software to provide the insights and data needed to design better architectures. **All** three first-generation RISC research projects involved compiler developments and significant language studies, before designing an architecture. Thus, a significant activity in our project is the exploration of approaches to parallel programming.

### 4.1 Parallel programming languages

We believe that there are **basically** three approaches to expressing **parallelism:**

1. Using conventional process structures and process communication techniques, such as those used in operating system design. Separate processes are specified and they synchronize and communicate by shared memory structures such as monitors or by message passing. This approach is most suitable for **very** large grain **parallelism.**
2. Extracting **parallelism** from existing sequential languages. The best example of this work **has** been carried on by David **Kuck** at Illinois. This approach has been reasonably successful for scientific programs written in **Fortran;** whether this technique can be used on a wide set of the problems that we are interested in (e.g., switch-level simulation of integrated circuits) is an open question.
3. Expressing the parallelism using a language that has natural parallelism and/or parallelism directives. **Multilisp** and QLISP are both LISP extensions that have explicit parallel constructs; concurrent versions of **Smalltalk** have also been proposed. Single-assignment or applicative (or value-oriented) languages have both implicit parallelism and explicit parallelism directives.

Our approach has been to concentrate on the latter **-** namely, a single assignment language with both inherent and explicit parallelism. The advantages of such languages include:

- They are implicitly parallel; only explicit data dependency limits parallelism.
- Explicit parallelism directives are easily incorporated; in addition these operators (what **Backus** calls

combining forms) can be applied with arbitrary functions. That is, the language allows us to apply any function to an entire vector in parallel, knowing that the result is independent of the order of evaluation.

- The absence of sideeffects means that any parallel evaluation that follows the explicit data dependencies will result in the same answer. This is not true for many parallel prog ramming extensions of conventional languages (e.g. it is not true in Multilisp).
- Similarly, because these languages are expression-oriented we can determine the set of values that must be communicated between two sections of the program. This means that sharing of data is determined at compile-time and that programs may be easily partitioned without any unforeseen communication requirements.

The challenges for such languages are significant The most obvious one is whether such languages will allow a wide range of applications to be written in such a way that significant parallelism can be exploited from them. Several groups (including our own)[10] are exploring this issue; the SISAL [8] research project has coded several significant problems in SISAL (a close relative of our language, SAL). The SISAL programs include:

- Simple - the LLL CFD benchmark
- RSIM - a version of Terman's switch-level simulator
- IV - a circuit extractor and interconnect verifier
- SPLICE - the Berkeley multilevel simulator

We are exploring the available parallelism in these programs and others with our compiler system, which is shown in Figure 5. The SAL system allows programs to be compiled both for sequential and parallel machines.

A second challenge for these languages is to automate the partitioning of programs. Automatic partitioning is vital to allow us to compare the architectural-algorithmic fit for an application. With automatic partitioning we can easily examine the effect of the interconnection media, processor count, and overall communication versus computation capabilities of a machine. We discuss our approach to automatic partitioning in the next section.

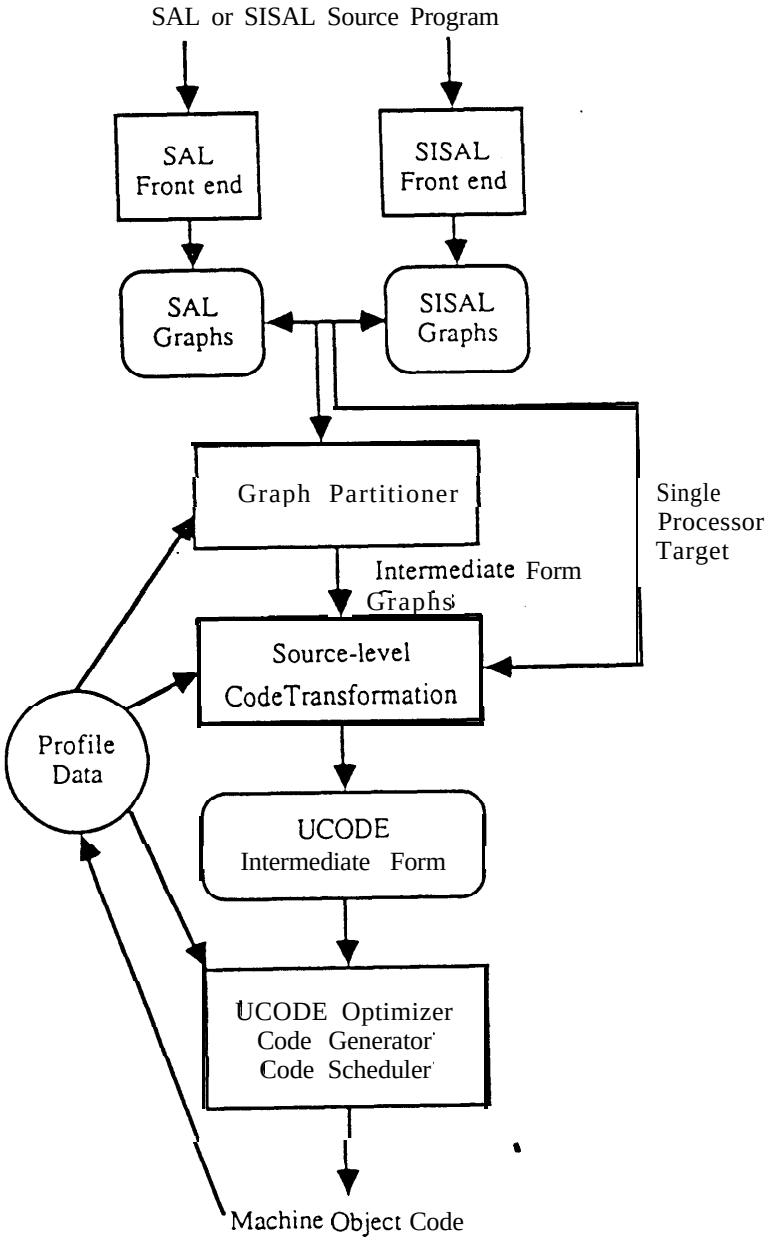
Because these languages are purely applicative, they follow a copy semantics; that is, any update of an object (structured or simple) must appear as if the object were copied and then updated. We discuss our solution to eliminating this overhead shortly.

## 4.2 Automatic partitioning

Our approach to partitioning is based on an approach often called macro dataflow. We aim to transform the program into a dataflow graph where each node represents a sequential computation and the arcs represent all the data dependencies among computations. Thus, at any time in the computation, any node whose predecessors have all been evaluated may be executed. The applicative, side-effect-free property of the language plays an important role

[10] There are many projects evaluating the potential parallelism in such an approach these include:the SISAL project (Lawrence Livermore, U. Colorado, U. Manchester, and Digital Equipment Corporation), the VAL and Id projects (MIT), Rediflow (U. Utah), and the graph reduction word at MCC.

Figure 5: The SAL Compilation System



in facilitating this decomposition. All data dependency is explicit in the program and no global shared memory with implicit aliasing exists. Furthermore, the computational model allows any subcomponent to be subdivided and represented as a composition of two functions.

Although a dataflow representation is used, there is no connection to a dataflow computation model as used in a dataflow architecture. Rather than represent individual instructions or operations, each node represents a grain; the granularity of the partitioned parallel program is defined by the size of the nodes. This partitioning process creates a graph where the node size is balanced by the amount of computation, the number of processors in the machine, the overhead to schedule a task, and the communication requirements of that node. Some of these factors may conflict; e.g., the optimal partitioning may use fewer processors than are available. The parameters describing the machine must supply the necessary input for the partitioning; we believe that a small set of key parameters can capture the important capabilities of a wide variety of machines.

We are exploring two approaches to parallel partitioning: a static, compile-time approach and a dynamic, runtime approach. In the static approach, we assume that the behavior of the program is well known and that an available execution profile provides a reasonable relative measure of both the execution time of various subcomputations as well as their frequency". With this assumption, our goal is to partition the program into a set of tasks and to schedule the tasks on processors at compile-time. This completely eliminates the need for runtime scheduling, although synchronization among tasks is needed for data communication. This approach is discussed and explored in [11].

When such accurate information is not available or the relative costs or frequencies are heavily data dependent, we attempt only to partition the program at compile-time and rely on runtime assignment of tasks to processors. This partitioning attempts to obtain sufficient parallelism to keep all processors busy, but may choose to combine potentially separate computations when the communication cost to split them is too high. This approach is refined and evaluated in [12].

These partitioning algorithms are designed to be usable in a multiple machine environment. One set of inputs describes the characteristics of the machine: computational speed and communication cost (latency, overhead to initiate, bandwidth considerations). Since the output of this partitioning process is a dataflow graph that is compiled into our common intermediate format., we can relatively easily host this partitioner onto new architectures. Thus, we can experiment with automatic partitioning of a program to a variety of architectures, examining the algorithm-

---

"We are relying on the relative accuracy of this information; provided the execution costs and frequencies of components scale, the partitioning will be well chosen, except for boundary cases.

architecture match, as well as the suitability of this approach.

### 4.3 Optimization of expression-oriented programs

The basis of the MIPS compiler system (as well as the basis of the MIPS-X Pascal and C compilers) is the UCODE compiler. The components and flow of that compiler are shown in Figure 7. The global optimizer can do a wide-range of optimizations and global register allocation. However, applicative languages introduce a set of problems not well handled by these approaches. The copy-on-write problem is perhaps the most important of these; we would like to replace copies of updated objects, with update in place. In addition, the lack of side effects simplifies some optimizations, such as rangecheck elimination.

Why is the update-in-place optimization so important? The key reason is that any structure that is updated must be copied in a straightforward implementation to maintain the applicative semantics. In particular an iterative algorithm that basically updates portions of an array on each iteration, is required to copy the entire array on each iteration. The best example of this behavior occurs when we examine a single-assignment version of bubble sort. Because the array is updated on each iteration it must be copied (from the "old" value it has at the beginning of each iteration of the inner loop to the new value it has at the end). In some cases, simple analysis techniques can eliminate copies when the lifetimes of the source (often an old value) and destination (often a new value) do not overlap; although this optimization can be tricky because often the source becomes dead just as the destination becomes live. Unfortunately, more complex cases arise: in bubble sort the old and new values of the array overlap. Hence, a simple optimization will not work. In [5], we discuss optimization techniques that through the introduction of temporaries can optimize this and more difficult cases. For bubble sort, a straightforward implementation of an applicative program produces an $O(n^3)$ algorithm, while our optimization technique returns it to $O(n^2)$. Without these sort of optimization techniques the advantages of such languages for expressing parallelism may come to naught, because the inefficiency of these languages may dominate the speed-up obtained by parallelism.

## 5 Advanced Compiler Technology

Our compiler activities seek to directly support our architecture activities; we believe that the best new architectures are developed jointly with new compiler technology. This allows us to explore new tradeoffs between hardware and software technology. We discuss what the issues are in developing better compiler technology, what our approach is, and how to deal with languages such as LISP.

## 5.1 Problems and challenges

As we build higher performance RISC engines and boost the instruction throughput rate, it becomes increasingly difficult **to** maintain single-cycle execution (i.e. as close to one cycle per instruction as possible) and achieve the desired clock speed Hardware techniques can achieve high throughput on ALU operations, but it becomes very difficult to achieve single cycle execution on branches and memory references. Indeed some RISC machines, such as the Fairchild Clipper, do not even attempt to achieve singlecycle branches or memory references (these instructions take five or more cycles).

The deeper pipelining used in second generation RISC architectures (five stages in MIPS-X versus 2.5 in MIPS) makes it increasingly difficult to keep down branch delays and load delays. We use new compiler technology to reduce the cost of longer branch delays and to reduce the cost of load delays as well as the frequency of loads and stores.

A second issue that we are addressing is the behavior and performance of LISP. LISP is a very different language from the procedural languages (C and Pascal) that drove the design of the first generation of RISC engines. Our goal is to find out how different LISP is when measured by dynamic instruction usage, then to examine the most costly aspects of the language, and finally to determine what architectural enhancement might be appropriate for LISP and how effective they are.

## 5.2 LISP on a RISC (3)

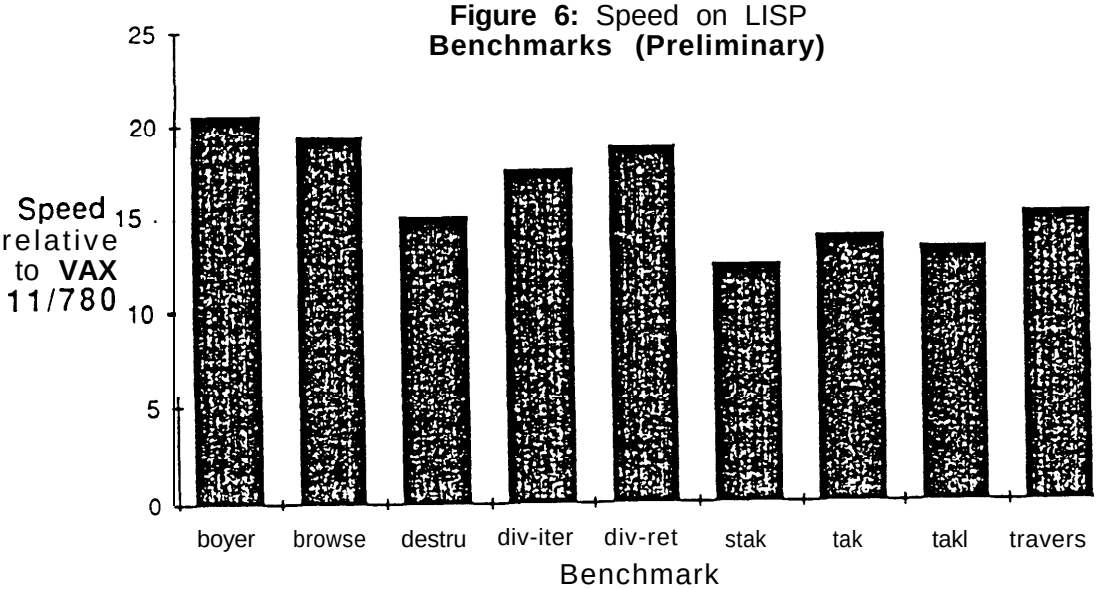
Our goals in this section of the project are threefold:

1. Characterize **LISP** behavior by measuring it on MIPS-X. We are evaluating how LISP is different as measured from the dynamic instruction mix and what various LISP features cost.
2. Explore better compiler technology for LISP, using our existing optimization technology. Topics under exploration include conventional and special register allocation, function integration, and conventional global optimizations.
3. Explore and evaluate architectural extensions to MIPS-X designed to improve the machine as a LISP engine.

Our current activities of LISP are based on PSL; they began before the Common LISP compiler became widely available. Nonetheless, the LISP systems are very similar and most observations of behavior and performance for PSL carry over to Common LISP. Our long-term plan is to initiate a Common Lisp port

Before we discuss our main research objectives in detail, let's look at how good a LISP machine MIPS-X is. Figure 6 compares the performance of LISP using Gabriel's benchmarks [1] on a VAX 1 1/780 (normalized to one in the plot) and MIPS-X. Both machines use the PSL compiler, and the programs were compiled without type checking on arithmetic and vector operations..

**Figure 6:** Speed on LISP **Benchmarks (Preliminary)**



To understand LISP behavior we are examining the low-level dynamic instruction behavior of LISP (see [13] for a full discussion). Since MIPS-X instructions are very basic operations, the MIPS-X instruction mixes show the basic behavior of LISP: is it memory access intensive, is it ALU intensive? As expected LISP programs tend to have higher frequencies of memory reference instructions; perhaps surprisingly, they also have higher frequencies of branches. These two classes of instructions are more costly and harder to make fast than ALU instructions.

We are also examining LISP behavior from a top-down viewpoint: what are the costs of the frequent LISP operations. The singlecycle nature of the architecture simplifies this study since the cost of a feature is easily measured by instruction counts. We are measuring a variety of operations including: function call cost, cost of LISP primitives versus user instructions, and the cost and frequency of list and vector operations.

We have developed some compiler techniques targeted directly at LISP; the high frequency and cost of procedure call emphasizes the need for effective interprocedural register allocation and procedure inlining (or register window hardware support).

We are also examining what specific architectural extensions might make sense and how much they can help. Such extensions might include a range of tag operations (checking, comparison, removal, and insertion), and function invocation support

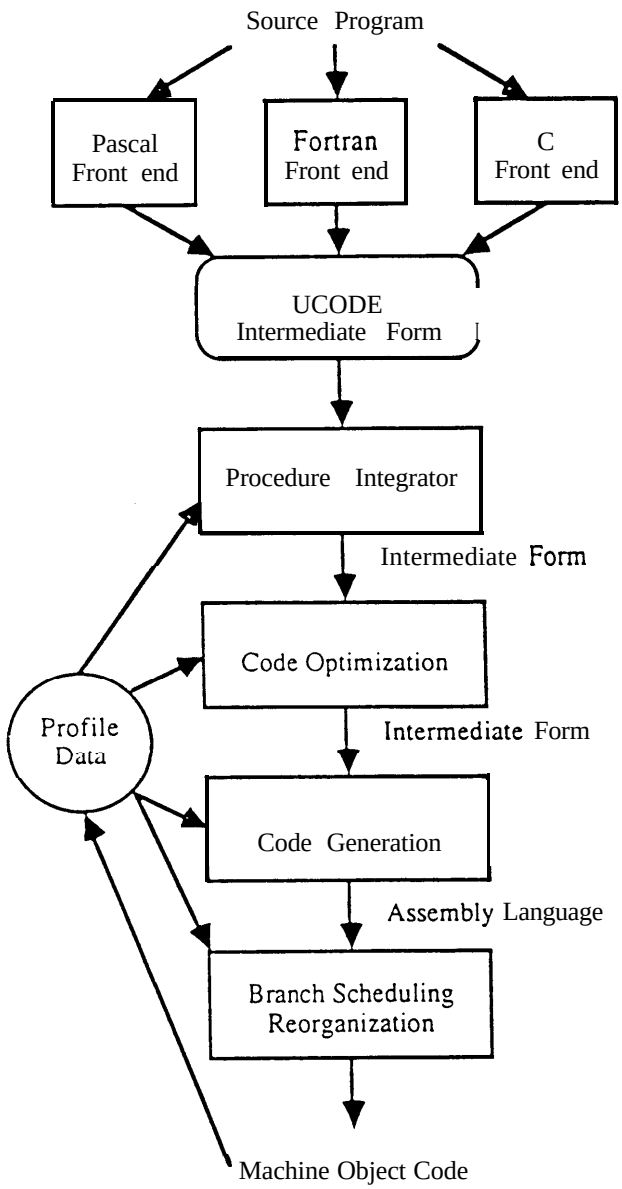
### 5.3 Profile-based compilation

Our compiler activities are aimed at reducing the cost and/or frequency of the most costly instructions: loads/stores and branches. The central theme of our activity is profile-based compilation. Each run of a program under development can produce a profile that is used to tune the program by optimizations using that profile. The automation of this process and its stability are also being explored. Figure 7 shows an overview of the compiler system.

A major component of our activities in this area is exploring the use of profiles in instruction scheduling. Instruction scheduling is needed to maintain single-cycle execution rates for instructions whose natural latency is long due to memory accessing constraints (branches and memory references are in this class) or due to deeper pipelining of the instruction (floating point operations are in this class). Branches have been the primary focus of our work and we have shown [7] that software branch delay scheduling is competitive with the most advanced hardware branch delay schemes used in very large mainframes. In addition, our techniques can be used to do load delay and floating point delay scheduling with a technique similar to, but simpler than, trace scheduling [4].

Loads and stores account for a significant fraction of the instruction mix, and hence are a major performance

Figure 7: The MIPS UCODE Compiler



limitation. Our present compiler technology is effective at eliminating the cheaper ALU instructions, and the branch counts are fixed by the initial program (and are single-cycle because of our effective scheduling of their delays). Hence, load/store instructions are the most opportune target for improving performance.

Our coloring-based register allocation algorithms are fairly effective, but are limited by the accuracy of the estimates on usage and by the high frequency of procedure calls. Even when procedure calls are not that frequent, register allocation is limited by the number of memory instructions used to save and restore registers at the calls. This cost is significant: on the **VAX,** the CALLS instruction (which saves and restores registers and does a procedure call) generates the most memory traffic of any instruction! Thus, we are focusing on two key problems: procedure integration and interprocedural register allocation.

Procedure **integration** is the optimization of selectively inlining procedures. When a procedure is integrated (i.e. the call is replaced with the body), several types of improvement in the runtime of the program can occur:

- The procedure call overhead, consisting of saving and restoring registers, setting up (and removing) the new activation record, and passing parameters (and return results) can all be eliminated
- The procedure integration allows conventional data flow analysis to analyze across the procedure so **that** information **about** aliasing is sharpened In this sense, procedure integration makes the data flow **analysis interprocedural, at least for this call.**
- Optimizations **across procedure call boundaries can be done; for** example, code motion can be done out of an integrated procedure whose call was nested in a loop.

Each integration of **a** procedure called more than once, increases the code size of a program This reduction in locality increases the cache miss ratio, thus substantially affecting the performance of the machine. In addition, common heuristics are not very effective in choosing which procedures to integrate. Frequently, there are two nearly identical calls to a procedure, one used frequently, the other used infrequently, perhaps in error situations. We have found that profile-based procedure integration can achieve results that are very close to those achievable by heuristics that double or triple the code size, yet profile-based integration may yield only a 10% to 25% growth in code size. Our current efforts are focused on getting an accurate space-time tradeoff model for procedure integration that accounts for cache effects.

Our other attack on load/store frequency is based on interprocedural register allocation. The idea behind inter-procedural register allocation is quite simple. As Patterson [9, 10] and others have observed, each procedure uses only a small number of registers. Hence, if we could allocate the registers of a calling and a called procedure into nonoverlapping parts of the register set, we could eliminate save/restore overhead for that pair. The complication comes from the nontrivial call pattern - the call graph is (ignoring recursion) a dag, not a tree. A procedure has a chain of ancestors, whose registers it must not use; furthermore, the procedure can have multiple callers, whose registers must all be allocated correctly to avoid the overhead (likewise for the ancestors of the

callers). Given profiling information, further optimization is possible. A large program will require saves and restores because the call chains are easily deep enough to use all the registers. With profile information we can attempt to **save** and restore registers at the cheapest (i.e. least frequently executed) locations.

David Wall at DEC Western Research Laboratory has developed an interprocedural allocation scheme based on a link-time algorithm We are basing our approach on a compile-time algorithm that will aim for higher efficiency, both by better allocation, and by using profile data to minimize the cost of the necessary saves and restores.

## 6 Conclusions

The MIPS-X-MP project involves four closely related efforts. **We** believe that real progress in computer architecture demands intimate knowledge of the technology (in our case, VLSI) and simultaneous software research efforts. By combining powerful processors together and providing **a software environment that can find** and schedule parallelism, **we hope to have a multiprocessor that is suitable for a** wide-range of applications.

# References

**1.** Gabriel, R. *Performance and Evaluation of LISP System.* MIT Press, Cambridge, Mass., 1985.

2. Agarwal, A., Sites, R., Horowitz, M. ATUM: a new technique for capturing address traces using microcode. Proc. 13th Sym. on Computer Architecture, IEEE/ACM, Tokyo, Japan, June, 1986.

3. Cheriton, Slavenburg, Boyle. Software controlled caches in the Nebula multiprocessor. Proc. 13th Sym. on Computer Architecture, IEEE/ACM, Tokyo, Japan, June, 1986.

4. Fisher, J., Ellis, J., Ruttenberg, J., Nicoiau, A. Parallel Processing: A Smart Compiler and a Dumb Machine. Proc. '84 Syrn. on Compiler Construction, ACM, Montreal, Canada, June, 1984, pp. 37-47.

5. Gopinath, K., Hennessy, J. Copy Optimization in Single-Assignment Languages. Working paper.

6. Gupta, A., Forgy, C. Measurements on production systems. CMU-CS-83-167, Department of Computer Science, CMU, 1983.

7. McFarling, S., Hennessy, J. Reducing the Cost of Branches. Proc. 13th Sym. on Computer Architecture, IEEE/ACM, Tokyo, Japan, June, 1986.

8. McGraw, J. et al. SISAL: Streams and Iterations in a Single Assignment Language. Language Reference manual M-146, LLNL, March, 1985.

9. Patterson, D. "Reduced Instruction Set Computers". *CACM 28*, 1 (January 1985), 8-21.

10. Patterson, D.A., Sequin, CH. "A VLSI RISC". *Computer 15*, 9 (September 1982), 8-22.

**11.** Sarkar, V., Hennessy, J. Compile-time Partitioning and Scheduling of Parallel Programs. Sym on Compiler Construction, ACM, Palo Alto, Ca., June, 1986.

12. Sarkar, V. and Hennessy, J. Partitioning Parallel Programs for Macro-Dataflow. Sym on LISP and Functional Programming, ACM, Boston, Mass., August, 1986.

13. SteenKiste, P. and Hennessy, J. LISP on a Reduced-Instruction-Set Processor. Sym. on LISP and Functional Programming, ACM, Boston, Mass., August, 1986.

14. Taylor, G., Hilfinger, P., Larus, Zorn, Patterson. Evaluation of the SPUR LISP Architecture. Proc. Sym. 13th on Computer Architecture, IEEE/ACM, Tokyo, Japan, June, 1986.

15. Wood, Eggers, Gibson, Hill, Pendelton, Ritchie, Taylor, Katz, and Patterson. An in-cache address translation mechanism Proc. 13th Sym on Computer Architecture, IEEE/ACM, June, 1986.

# Project Publications

1. Agarwal, A., Sites, R., Horowitz, M. ATUM: a new technique for capturing address traces using microcode. Proc. 13th Sym. on Computer Architecture, IEEE/ACM, Tokyo, Japan, June, 1986.

2. Chow, P., Horowitz, M. The MIPS-X Microprocessor. Proc. WESCON 85, IEEE, San Francisco, 1985.

3. Chow, P. MIPS-X Instruction Set and Programmer's Manual. Technical Report 85-289, Computer Systems Laboratory, Stanford University, October, 1985.

4. Gopinath, K., Hennessy, J. Copy Optimization in Single-Assignment Languages. Working paper.

5. McFarling, S., Hennessy, J. Reducing the Cost of Branches. Proc. 13th Sym on Computer Architecture, IEEE/ACM, Tokyo, Japan, June, 1986.

6. Sarkar, V., Hennessy, J. Compile-time Partitioning and Scheduling of Parallel Programs. Sym. on Compiler Construction, ACM, Palo Alto, Ca., June, 1986.

7. Sarkar, V. and Hennessy, J. Partitioning Parallel Programs for Macro-Dataflow. Sym on LISP and Functional Programming, ACM, Boston, Mass., August, 1986.

8. SteenKiste, P. and Hennessy, J. LISP on a Reduced-Instruction-Set Processor. Sym. on LISP and Functional Programming, ACM, Boston, Mass., August, 1986.

# Project Members

| | |
|---|---|
| John Acken | MIPS-X I-cache implementation |
| Anant Agarwal | Cache studies |
| Jim Celoni, S J. | SAL implementation |
| Y. cho | Single-assignment language compiler project |
| Paul Chow | MIPS-X |
| C.Y. chu | Mils and Mils-x instruction level simulators for MIPS and MIPS-X |
| M. Ganapathi | Compiler system: code generation and interprocedural analysis |
| K. Gopinath | High-level optimization of single-assignment programs |
| Glenn Gulak | MIPS-X functional simulator |
| Mark Horowitz | MIPS-X Project Leader |
| Scott McFarling | Profile-based branch scheduling, reorganization, and procedure integration |
| Steven Przybylski | Resident MIPS fellow |
| Arturo saltz | MIPS-X |
| Vivek Sarkar | Program partitioning for parallel architectures |
| Richard Simoni | MIPS-X implementation |
| Jeff Sprouse | MIPS-X board design |
| Don Stark | MIPS-X implementation |
| Peter Steenkiste | LISP compiler for MIPS-X |
| Steve Tjiang | MIPS-X compiler |
| Malcolm Wing | New version of UOPT; interprocedural register allocation |