MPOC: A Chip Multiprocessor for Embedded Systems

Stephen Richardson
Internet Systems and Storage Laboratory
HP Laboratories Palo Alto
HPL-2002-186
July 8th , 2002*

CMP, chip multiprocessor, embedded processor, embedded DRAM, CPACM

MPOC was an ambitious microprocessor project undertaken at Hewlett Packard's Palo Alto Research Lab from about 1998 until early in the year 2001. MPOC was designed to be a single-chip community of identical high-speed RISC processors surrounding a large common storage area. Each processor would have its own clock, cache and program counter. Each processor was to be small and simple, such that it would run very fast using minimal power requirements.

MPOC attempted to break new ground in several categories, including 1) novel funding for microprocessor research; 2) introducing multiprocessing to the embedded market; 3) trading design complexity for coarse grain parallelism; 4) a novel four-stage microprocessor pipeline; and 5) using co-resident on-chip DRAM to supply chip multiprocessor memory needs.

MPOC's first generation design targeted the embedded printer market. Using TSMC's 0.18 micron CMOS process including combined logic and DRAM, it planned to place four processors on a single chip. Each processor would be 1.5 x 2 mm, including a 4KB instruction cache and a 4KB data cache. The processors shared a single on-chip 4MB DRAM occupying about 4 x 8 mm, designed to fill a 32B cache line in 20 ns across a 256b internal bus. Total chip size would be approximately 55 sq mm with a target sell price of $40 to $50. An alternative 0.13 micron part could put 4 CPU's and 12 MB of DRAM on a 59 sq mm chip.

* Internal Accession Date Only

MPOC: A Chip Multiprocessor for Embedded Systems

Stephen Richardson, HP Laboratories, June 2002

This document chronicles the story of MPOC, an ambitious microprocessor project undertaken at Hewlett Packard's Palo Alto Research Lab from about 1998 until early in the year 2001. MPOC was designed to be a single-chip community of identical high-speed RISC processors surrounding a large common storage area. Each processor would have its own clock, cache and program counter, all operating synchronously. Each processor was to be small and simple, such that it would run very fast using minimal power requirements.

MPOC attempted to break new ground in several categories, including

- novel funding for microprocessor research;
- introducing multiprocessing to the embedded market;
- trading design complexity for coarse grain parallelism;
- a novel four-stage microprocessor pipeline;
- using co-resident on-chip DRAM to supply chip multiprocessor memory needs.

1. Novel funding for microprocessor research

Like Compaq's Piranha project [Barr00], the MPOC team's ultimate goal was to produce large scale multiprocessors for enterprise and technical computing. Looking far ahead into the future of Moore's Law and massive server consolidation in the data center, we envisioned a time when there would be a market for hundreds of processors on a single chip. Each processor would serve as a separate independent web server, or could perhaps be part of a highly efficient database cluster, or perhaps even participate in network packet switching and routing.

We realized up front that such an ambitious undertaking would require massive resources in terms of people and money, and that there might be novel ways to reduce the risk. In that spirit, we decided to target a midway point that could, itself, become a revenue-generating product. This midway point would establish a proof of concept, while at the same time initiate an income stream that we could use to further bootstrap the ongoing research (see Figure 1).

The intermediate point we chose for the investigation placed four small processors on a chip with embedded DRAM. This vehicle would allow us to investigate and begin to work through initial challenges with respect to applications, operating systems, and exotic processes necessary for the DRAM. For the revenue generation part of the plan, we discovered a potentially lucrative customer within HP itself, through which we hoped to develop a processor that would have general applicability outside HP as well. The four-processor solution was dubbed MPOC, for "Many Processors, One Chip." The follow-on project was to be HPOC, or "Hundred Processors, One Chip."

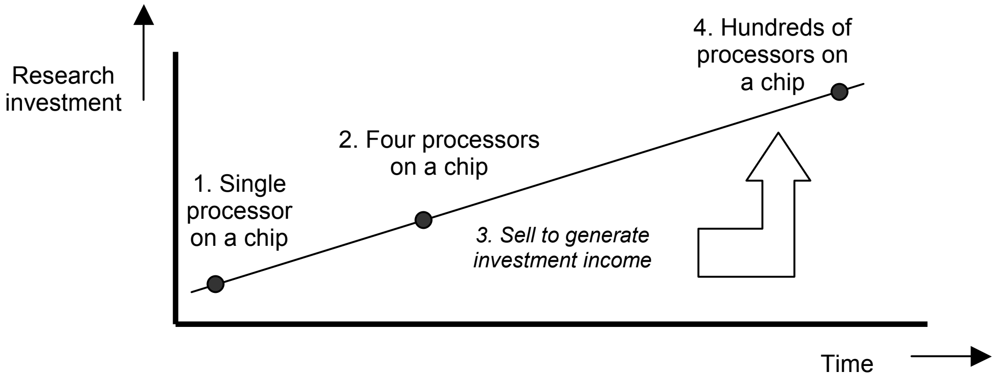

Research investment
4. Hundreds of processors on a chip
2. Four processors on a chip
1. Single processor on a chip
3. Sell to generate investment income
Time

Figure 1: Financing an ambitious long-range research project.

2. Introducing multiprocessing to the embedded market.

The MPOC team had had previous experience helping with a project that tried to introduce VLIW processing into the embedded market, specifically for printing. A large barrier to the success of this earlier project was the many thousands of lines of C++ code that had evolved over many years to run on a specific processor and operating system. Changing to a new processor would involve an expensive port using tools and compilers that had, at that time, not undergone extensive testing in an industrial setting. For this and other reasons, the printer group decided not to use the VLIW part.[1]

Nonetheless, the lucrative printer market was still looking for a significant boost in image processing speed. A quick study found low-hanging fruit, in the form of coarse grain parallelism, in a core part of the image processing algorithm. Better still, the parallelism could be extracted with only minimal code change. The initial port of this code from sequential to parallel form took one team member about two weeks to perform, and consisted of the modification of a very few of the over 20,000 lines of C++ code in the application. At the highest level, parallelization of the code was conceptually a matter of transforming the sequential outer loop

```
while (inputExists) {
      DoNextStrip(stdin);
}
```

into a fully parallel loop

```
while (inputExists) {
      strip s = readNextStrip(stdin);
      FORKPROCESS {
            DoNextStrip(s);
      }
}
```

Parallel execution showed a speedup of 3.4x on four processors, and more than 7x on an eight-processor machine. This was not simulated speedup, but actual speedup achieved

[1] A follow-on to the original VLIW processor found success in the embedded market as ST Microelectronics' LX processor core [Fara00][Cata01].

by running the modified benchmark on a four-processor HP9000 server running HP-UX 11.0. The achievement seemed remarkable (and lucky, as well) given the small amount of effort that was spent understanding and transforming the original code. It seemed probable that further study could only improve the speedup.

HP's printer group showed extreme interest in this work, and went so far as to say they would definitely use this part if it could be built at target speed, cost and delivery date.

3. Trading design complexity for coarse grain parallelism

MPOC's philosophy drove its design team to eschew the tradeoff, commonly made among modern architects, that uses an increasingly large area of processor silicon for an increasingly small advantage in instruction level parallelism (ILP). To this end, the MPOC processor core was designed to be not 4-, 3- or even 2-way superscalar, but to execute a single in-order stream of instructions in simple scalar mode. By making this core as small as possible, it would be possible eventually to fit many more of the processors on a single die, and to achieve a degree of coarse grain parallelism that would more than compensate for the small loss in ILP.

To better illustrate this tradeoff, suppose we have a parallel application with an ordinary degree of instruction level parallelism. The following alternatives might reasonably be considered for a single-chip processor to power the application:

- a die with a single huge processor might achieve 1.8 IPC (instructions per cycle), for a speedup of 1.8x versus a baseline smaller processor having 1.0 IPC;

- a die containing four large processors, each achieving 1.5 IPC, might attain a speedup of about (4 x 1.5) or 6x versus the baseline;

- a die with eight small processors, each achieving 0.9 IPC, might attain a speedup of about (8 x 0.9) or 7.2x versus the baseline.

Given the alternatives for this example, it becomes clear that more, smaller processors will outperform fewer, larger processors. In addition, the small processors are easier to test, design and build than the larger processor. Finally, because the processors operate independently of one another, the smaller processors should result in shorter global signal lines and thus should be able to achieve a higher clock frequency than larger processors.

The processor core chosen for MPOC was a simple scalar MIPS processor, designed to closely match the needs of the embedded market in general, where MIPS predominates, as well as to provide the easiest possible transition path for the specifically targeted printer customers. To better tailor the core to our specific needs, we decided to design the processor ourselves, within HP's research lab. The processor had a novel (for its extreme simplicity) four-stage pipeline, resulting in a branch penalty so small that it obviated the need for complicated branch prediction schemes.

4. The MPOC four-stage pipeline

Flaunting convention, MPOC used a four-stage pipeline to calculate computational results. Ordinary processors use at least five stages [Henn90]:

- an F-stage to fetch the instruction from the instruction cache,
- a D-stage to decode the instruction,
- an E-stage to calculate arithmetic results and/or a memory address,
- an M-stage during which the processor can access the data cache; and
- a W-stage during which the processor writes results back to its register file.

Modern workstation-class processors often have many more stages, including an additional R stage to handle the complexity associated with the register file and bypass mux of deeply pipelined superscalar, out-of-order and/or VLIW designs. Such designs often have register files with ten to twenty write ports and fifteen to twenty read ports.

MPOC managed to get away with four stages, eliminating the M stage, for two reasons: 1) the small first level caches can be accessed in a single cycle and 2) the simple base-plus-offset addressing scheme of the MIPS instruction set allowed addresses to be calculated in the second half of the D stage. MPOC's simple scalar design allowed its register file to have only one write port and three read ports.

4.1 Loads and Stores

MIPS load and store instructions add a sign-extended sixteen-bit offset to the contents of a register to produce a memory address. The high sixteen bits of the offset will be either all zeroes or all ones, depending on the sign of the offset. The address calculation can thus be implemented as a sixteen bit adder for the low sixteen bits, and a four-input multiplexer for the high sixteen bits, as seen in Figure 3.

The hit or miss signal for a data cache access does not appear until late in the E stage of the pipeline. In the case of a load, the data has already been fetched from the cache, but the miss signal can simultaneously halt the pipe and prevent the incorrect data from being written to the register until the correct data arrives, as shown in Figure 4. In the case of a store, the miss signal arrives in time to prevent incorrect data from going into the cache until the tags can be updated, dirty data can be written out to memory if necessary, and the pipe can be restarted. In Figure 4, as in all pipeline diagrams in this document, the letters F, D, E and W denote the Fetch, Decode, Execute and Write stages respectively. An x denotes a stall in the pipeline.

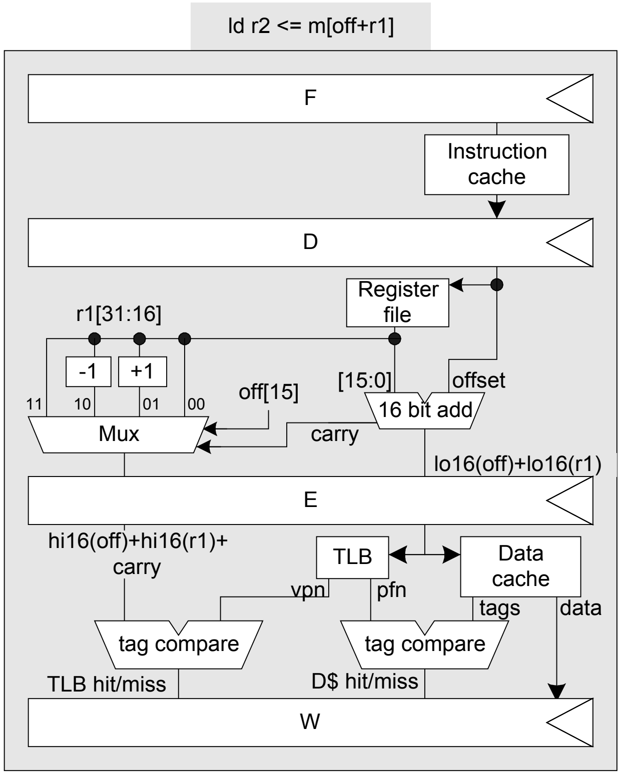


Figure 3: Accessing the data cache. Loads complete in E stage, stores overflow to W stage.

```
ld    F  D  E` x  …  x  W
ld+1     F  D  x  …  x  E  W
ld+2        F  x  …  x  D  E  W
ld+3                    F  D  E  W
```

Figure 4: Miss signal from data cache arrives in E stage, halts pipe.

```
st    F  D  E  Wa (store accesses dcache data in W stage)
ld       F  D  x    (bubble waiting for dcache data)
...               Ea W (load wants dcache data in its E stage)
ld+1        F  x  D  E  W
ld+2              F  D  E  W
```

Figure 5: Store followed by a load results in a single pipe bubble.

Loads access the data cache tags and data cache data in the E stage of the pipeline. Stores access the data cache tags in stage E and the data cache data in stage W. Thus, while the tag accesses pipeline perfectly, a store followed by a load results in a one-cycle pipe bubble, as shown in Figure 5.

Using the result of an arithmetic operation to calculate a memory address incurs a one-cycle penalty, because the arithmetic operation result is not available until the end of the E stage, and the memory address computation occurs during the D stage, as shown in Figure 6a.

Referring to Figure 6b, load data arriving at the end of the E stage is immediately available via register bypassing to a successive arithmetic use of that data.

```
add ->r1  F  D  E` W (result avail end of E stage)
ld (r1)      F  x `D  E  W (bypassed to ld D stage)
```

Figure 6a: One-cycle penalty if ALU op result used for address computation.

```
ld    F  D  E` W (loaded value avail end of E stage)
add      F  D `E  W (bypassed to add)
```

Figure 6b: Load followed by dependent use causes no pipe bubbles.

4.2 Branches

The MIPS instruction set mandates that the instruction following a branch is always executed, regardless of the direction eventually taken by a branch (with one exception, which we shall discuss later). This extra instruction allows the pipeline to calculate the target of the branch, so that it can speculatively fetch the target in the following cycle. This results in no penalty for taken branches and a one-cycle penalty for non-taken branches, as shown in Figures 7 and 8. The branch-likely instruction can, in the unlikely case, incur a further penalty. This instruction requires that the delay slot be nullified when the branch fails to go to its target, resulting in a two-cycle penalty, as illustrated in Figure 9.

```
br    F  D  E` W (cond avail at end of D stage)
br+1     F  D  E  W (delay slot)
targ        F  D  E  W (fetch target instruction)
```

Figure 7: No penalty for taken branch or taken branch-likely.

```
br    F  D  E` W (cond avail at end of D stage)
br+1     F  D  E  W (delay slot)
~~targ~~        F `x  x  x (fetch target instr (nullified))
br+2          `F  D  E  W (oops! fetch sequential instr)
```

Figure 8: One-cycle penalty for non-taken branch.

```
brL   F  D  E` W (cond avail at end of D stage)
~~brL+1~~    F  D `x  x (delay slot (nullified))
~~targ~~        F `x  x  x (fetch target instr (nullified))
brL+2         `F  D  E  W (oops! fetch sequential instr)
```

Figure 9: Two-cycle penalty for non-taken branch-likely.

Figure 10 shows a simplified diagram of the logic for the branch pipeline.

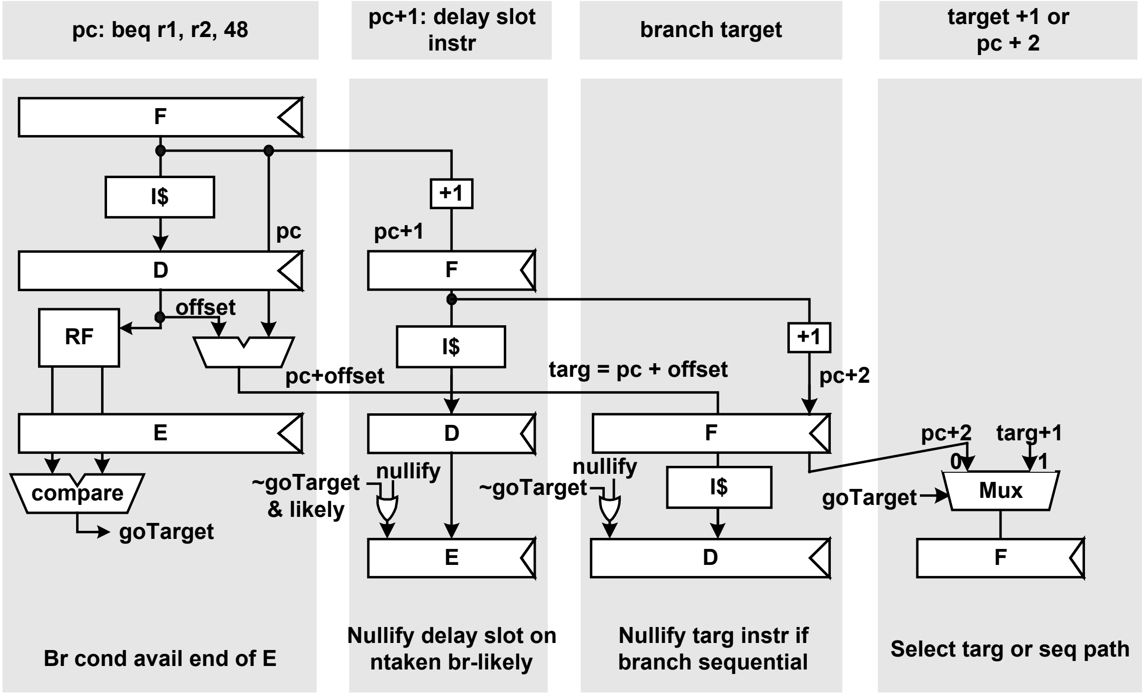


Figure 10: MPOC branch architecture.

4.3 Performance of the MPOC four-stage pipeline

Figure 11 summarizes the instruction penalties endemic to the MPOC four-stage pipeline. An early estimation of perfect-memory performance assumed no particular intelligence on the part of the compiler, and made the following assumptions: a store will immediately precede a load about five percent of the time; an address calculation will immediately precede a dependent memory operation about five percent of the time; branches account for about twenty percent of the dynamic instruction mix, about thirteen percent taken plus about seven percent untaken. (The branch-likely instruction is almost never used in general code.) These assumptions resulted in a predicted 0.85 IPC (instructions per cycle) on general code. Later measurements on benchmarks of interest showed an actual performance of 0.91 IPC, about ten percent better than predicted. These results are summarized in Figure 11 and elaborated more fully in Figure 12.

Interlock	Per instance	Estimated	Measured
st-ld	1 cy	0.05	0.04
ld-use	0 cy	0.00	0.00
addr-mem	1 cy	0.05	0.04
bt	0 cy	0.00	0.00
bnt	1 cy	0.07	0.02
bLt	0 cy	0.00	0.00
bLnt	2 cy	0.00	0.00
Total CPI		1.17	1.10
Instructions per cycle		**0.85**	**0.91**

Figure 11: Early estimate of .85 CPI was later measured as 0.91.

Figure 12 shows MPOC's actual performance on the benchmarks of interest. The benchmarks include code from SPEC benchmarks, printer and image processing code, Dhrystone and others. The pipeline executed, on average, about 91 instructions for every hundred processor cycles (0.91 IPC). IPC varied from as high as 0.95 for tetra down to 0.83 for ekx. Figure 13 shows the same information as a bar chart.

	CPI penalty				IPC
Benchmark	**Addr-Mem**	**Store-Load**	**Non-Taken**	**Total**	
tetra	.036	.009	.006	.051	0.95
bmark	.027	.019	.016	.062	0.94
trilinear	.041	.017	.006	.064	0.94
md	.000	.066	.000	.066	0.94
copymark	.013	.030	.026	.069	0.94
tinypeg	.024	.031	.017	.072	0.93
radial	.055	.009	.009	.073	0.93
des	.022	.032	.019	.073	0.93
testmontg	.051	.035	.004	.090	0.92
gs	.028	.031	.031	.090	0.92
mpeg	.010	.084	.012	.106	0.90
dhry	.048	.037	.028	.113	0.90
m88ksim	.061	.035	.019	.115	0.90
go	.063	.026	.029	.118	0.89
dmpt	.022	.064	.032	.118	0.89
cc	.064	.025	.038	.127	0.89
li	.099	.027	.042	.168	0.86
ekx	.081	.072	.047	.200	0.83
AVERAGE	***.041***	***.036***	***.021***	***.099***	***0.91***

Figure 12: Pipeline performance on benchmarks of interest.

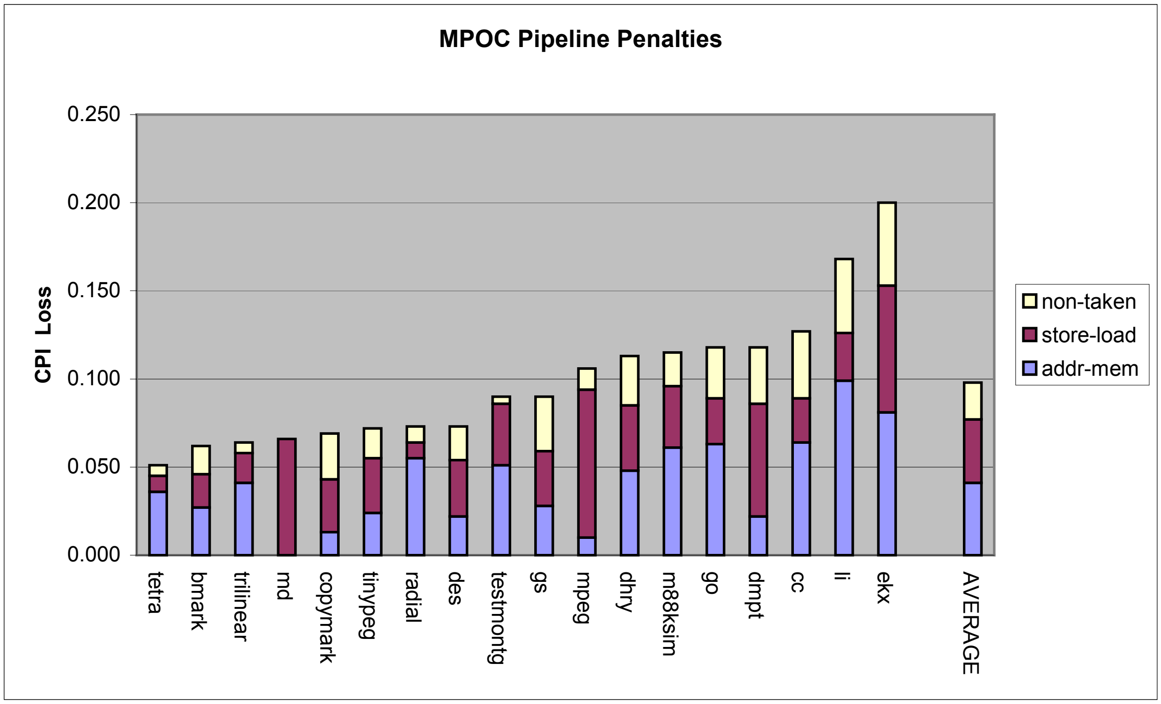
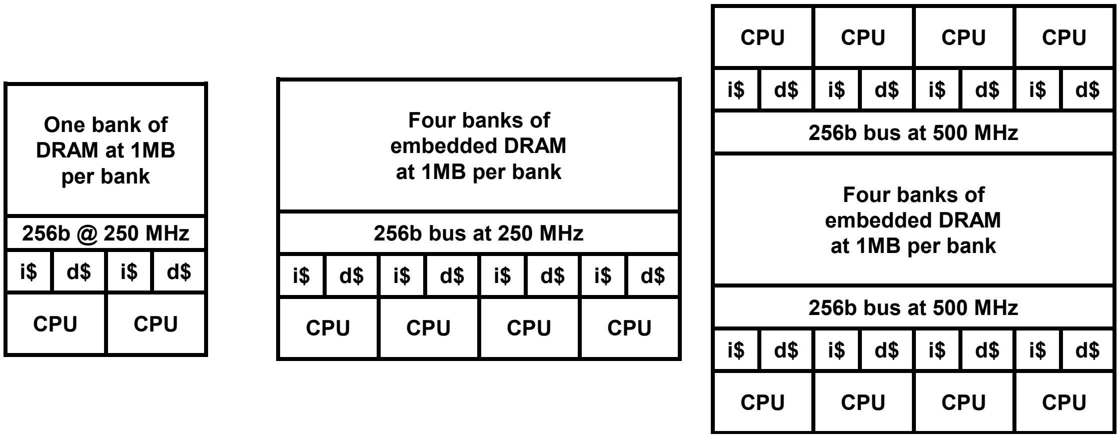


Figure 13: Pipeline performance on benchmarks of interest.

5. Using co-resident on-chip DRAM to supply CMP memory needs

MPOC's original plan would have placed 1MB to 4MB of DRAM on the same silicon die with the four processors, as shown in Figure 14. Later technology quotes indicated that it might be possible to place as much as 12 MB on a high end part.



Low end .25u part: Two CPU's at 240 MHz

Intermediate 0.18u solution: Four CPU's at 500 MHz each

High end deep-submicron solution: Eight CPU's at 1 GHz each

Figure 14: Design alternatives for a first generation MPOC family.

The embedded DRAM was by far the most technologically risky part of the project. Therefore, a back-up plan would allow us, up to the very last minute, to be able to replace this portion of the chip with a more standard SRAM solution. At its outset, the MPOC team counted among its strengths a certain amount of DRAM design expertise. It was hoped that this team could monitor and perhaps even shepherd the technology to a point where it could advantageously be used by the MPOC processor at the time of tape-out. Technology timing is always tricky, however, and, despite a promising start, a late report by a key team member remained inconclusive.

If Moore's law continues to hold, however, the desired amount of memory per chip will eventually become available, whether as SRAM or DRAM, and the question will still remain as to how best to use this massive *local* memory in a chip multiprocessor setting.

One way to manage the local memory would be as a traditional cache, organized as one or more sets of data lines. Each time a processor needs a data object, it checks the cache for the existence of a line containing the object. If the line is not there, the cache chooses a victim line. If the victim is dirty, the cache writes it out to remote memory, replacing it with the line from remote memory that holds the desired data object. This approach requires the cache to associate an address tag with each data line it holds, so it can tell an inquiring processor whether the line in cache matches the address of a requested data object.

Another way to manage the local memory would be to treat it as part of a unified physical address space that includes both local and remote memory. For example, the local memory might be assigned physical addresses 0 through `0x1fffff` while remote memory might be assigned physical addresses `0x200000` through `0xffffff`. Given such a layout, several schemes are possible to help minimize average memory latency. For instance, in a system without virtual memory translation, a program might be compiled and linked such that its most commonly used addresses reside in local memory, while less frequently used addresses live in remote memory.

Where virtual memory translation exists, the operating system can dynamically decide whether to map a given virtual page to local memory or to remote memory. The map can change on the fly and pages can migrate into and out of local memory from remote memory whenever the operating system chooses. Most probably the page migration would be triggered by some specific event such as a page fault from the translation lookaside buffer. In general, when a new page comes in from remote memory, some victim page must first be replaced in the local memory. If the victim is dirty, it must be written out to remote memory before the new page can come in. This scheme is similar to what one finds in a traditional non-uniform memory architecture (NUMA) or an I/O cache. It also resembles a traditional cache scheme, except where each cache line is the size of a memory page, and the tags reside in the TLB and in the data structures of the operating system.

An advantage to the unified physical address scheme is the elimination of the tag overhead in the local memory. A down side is the fact that it writes out an entire page on dirty-page cast out, regardless of how many lines in the page were actually dirty. Another down side is the fact that the local memory then brings in an entire page of data, regardless of how many lines actually get used. To help reduce the magnitude of these problems, we introduced CPACM, or Combined Paged And Cached Memory.

CPACM reduces the Write Wait associated with writing out a dirty page from local to remote memory and the Read Wait associated with reading a new page in to local memory from remote memory. CPACM associates a valid bit and a dirty bit with each line of each page in local memory. Each time a processor writes to a line in local memory, it also sets the dirty bit associated with that line. When the operating system wants to bring a new page into local memory, it first chooses a victim page as usual. Instead of writing out the entire victim page, however, it writes out only those lines whose dirty bit has been set. Then it clears the valid bits for all lines in the cache, bringing in and marking as valid *only the missing data line*. Finally, the operating system updates its page table (and the TLB) to reflect the new virtual to physical mapping. When a processor needs a data object from the new page, it first checks the valid bit for that line, bringing it in from remote memory if needed.

The CPACM scheme is similar to the technique of subblocking in ordinary caches, where each block has a tag and each subblock has its own valid and/or dirty bit. By making the block the same size as a page and using the existing virtual memory system, however, CPACM avoids the overhead of per-block tags in the memory. Furthermore, by using its ability to map any given virtual page to any given physical page in local memory, CPACM achieves the performance of a fully-associative cache while retaining the speed benefits of having a direct-mapped cache. Combined Paged and Cached Memory is described more fully in a technical report available from HP labs [Kelt00].

6. MPOC Status

MPOC's original project schedule is shown below as Figure 15. The plan called for a two year initial design period, followed by a second phase during which the project would be passed on to a design partner for industrial development.

By the beginning of the year 2001, when the MPOC project started to ramp down, several key technology decisions were in place:

- Process: TSMC 0.18 micron CMOS using combined logic and DRAM process.
- Floorplan: Four processors on chip, each 1.5 x 2 mm, each including a 4KB instruction cache and a 4KB data cache.
- On-chip main memory: One central 4MB DRAM occupying about 4 x 8 mm, designed to fill a 32B cache line in 20 ns across a 256b internal bus.
- Price: Total chip size approximately 55 sq mm with a target price of $40 to $50.
- Alternative 0.13 micron part could put 4 CPU's and 12 MB of DRAM on a 59 sq mm chip.

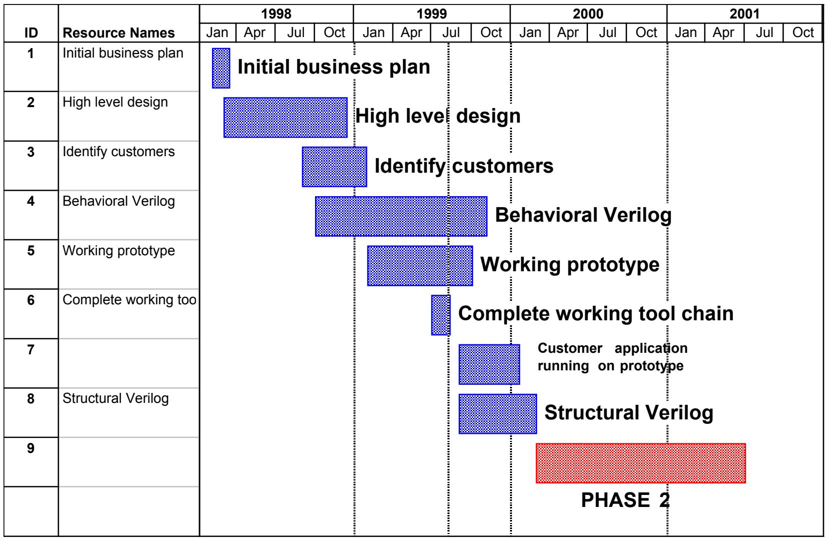


Figure 15: MPOC's original schedule.

The team built a working prototype of the MPOC system, consisting of four single-processor motherboards connected to a VME backplane, with a separate board for memory. The prototype used MIPS R4700 processors. It included a complete tool chain with compiler, debugger and multi-threaded operating system, all based on ISI's (now Wind River's) pSOS. The customer's core print application had been ported to and was up and running on the prototype. On this and other applications, MPOC showed significantly higher performance than competitive offerings, while using lower power and less die area.

Figure 16 shows the final status of the behavioral and structural Verilog, both of which were developed in parallel using MicroMagic EDA tools. The Verilog design for the processor was mostly complete except for the on-chip memory interface and the exception unit. HSPICE simulations of critical paths in TSMC's 0.18 micron process showed a worst case of about 2 ns, limited by the speed of our untuned adder.

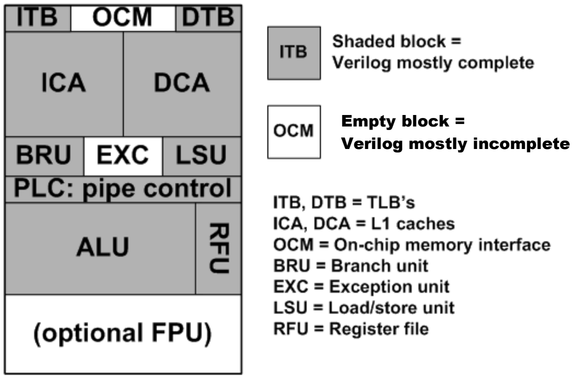


Figure 16: Final status of the Verilog design.

Phase 1 of MPOC's original schedule (business plan, design, Verilog, etcetera) was largely complete by the late summer of 2000. Phase 2 consisted of the attempt to find a manufacturer and industrial design team that would bring the part to market. The preferred path would have been to transfer the design to HP's internal team at Fort Collins. Fort Collins considered the design as a candidate for its New Business Creation program (NBC), but in the end the NBC program itself was dropped.

At this point, the design team began to transition over to new and follow-on projects, while a background process continued to search for partners externally. In May 2001, HP Labs made the strategic decision to abandon hardware research, and the team disbanded. The MPOC prototype, along with the Verilog design of the core processor, was donated to Stanford's Hydra research group.

Acknowledgments

The following people comprised the MPOC team. Each person made key contributions to the project, and they all worked together to create something that, together, was much greater than the sum of its parts.

- Stuart Siu, circuit design
- Stephen Richardson, logic design and project management
- Gary Vondran, Project Lead and logic design
- Paul Keltcher, processor simulation, application development
- Shankar Venkataraman, OS and application development
- Krishnan Venkitakrishnan, logic design and bus architecture
- Joseph Ku, circuit design
- Manohar Prabhu and Ayodele Embry, interns

The project is dedicated to the memory of Gene Emerson, without whose guidance and encouragement nothing would have been accomplished.

References

[Barr00] L.A. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets and B. Verghese. "Piranha: A Scalable Architecture Based on Single-Chip Multiprocessing", in *Proceedings of the 27th International Symposium on Computer Architecture*, June 2000.

[Cata01] Anthony Cataldo. "HP helps put VLIW into STMicro's Processing", EE Times, 19 December 2001, http://www.eetimes.com/story/OEG20011219S0038.

[Fara00] Paolo Faraboschi, Geoffrey Brown, Joseph A. Fisher, Giuseepe Desoli and Fred Homewood. "Lx: A Technology Platform for customizable VLIW Embedded Processing", in *Proceedings of the 27th International Symposium on Computer Architecture*, June 2000.

[Henn00] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach,* Morgan Kaufmann Publishers, Inc., San Mateo, California.

[Kelt00] Paul Keltcher and Stephen Richardson. "CPACM: A New Embedded Memory Architecture Proposal", HP Labs Technical Report HPL-2000-153, 21 November 2000.