

---

# A SINGLE-CHIP MULTIPROCESSOR FOR SMART TERMINALS

---

MERLOT, THE FIRST MP98 ARCHITECTURE PROTOTYPE, PROMISES 1-GIPS PERFORMANCE AT 1 WATT FOR 1.3-V OPERATIONS IN SUPPORT OF SMART 21ST-CENTURY INFORMATION TERMINALS.

..... What will information terminals look like in the 21st century? We believe they will require three important features: mobility, intelligence, and diversity. With the Internet, a large number of our daily activities will take place on information terminals. Mobility increases their convenience; for instance, wherever we are, we bring global currency with electronic commerce, and whenever we want to go, we are in public or business offices with electronic government and virtual offices. Intelligence should be another key issue in the new millennium. Predictions indicate that many artificial objects will acquire intelligence such as “automobiles that understand human speech, artificial bugs that fly, robots with human-like emotion, ...”<sup>1</sup>

Our information terminal will be a sensible partner as well as a smart interface to the digital world. The terminal will understand what we say, help us communicate with people speaking any language, and at the end of the 21st century might talk with creatures. Since everyone will have her/his own terminal for different times, places, and occasions, an enormous variety of terminals will be produced. This will result in a diversity of terminals and public servers should answer all requests from these terminals.

To attain these features, low-power and high-performance microprocessors are indispensable. Obviously, lower power enhances

mobility by enabling longer battery lives. High performance is necessary for intelligence and diversity. The artificial intelligence needed for information terminals includes recognition and understanding at first, and imagination and creation later in the middle of the century. For “true” intelligence, we require much higher computer performance, especially for dictionary handling and searching. For example, about 30% of CPU time is spent in a speech recognition system, as we will show later. At this ratio we greatly increase our ability to analyze the experiences of the speaker.<sup>2</sup>

The diversity of terminals demands standardization of information descriptions because huge varieties of terminals will access the information. Although XML and Java are currently becoming artificial-language standards, language abstraction continues unabated. Interpretation of such high-level abstract languages requires high performance.

The MP98 low-power, high-performance microprocessor architecture supports smart information terminals using the technologies of single-chip multiprocessor and low-voltage circuits. Its first prototype (code named Merlot) achieves 1 giga instructions/sec (GIPS) at 1 watt.<sup>3-5</sup> In this article, we briefly describe the MP98 architecture from a viewpoint of software. Then, we realize several basic algorithms in artificial intelligence on our multithreaded

**Masato Edahiro**  
**Satoshi Matsushita**  
**Masakazu Yamashina**  
**Naoki Nishi**  
**NEC Corporation**

architecture, and present performance estimates for a speech recognition application. We focus on intelligence because we expect it to be the most performance-demanding aspect of future information terminals.

### A single-chip multiprocessor

MP98<sup>4,6</sup> achieves high performance as well as low power with a single-chip parallel architecture. Since Merlot issues eight instructions per clock (four processing elements of two-way superscalar instructions), it needs only 125 MHz for its 1-GIPS peak performance. Merlot uses 1.3 V of supply voltage, and power dissipation is about 1 W for this level of performance. On the other hand, for applications that require low performance, MP98 reduces the number of working processing elements, which saves power dissipation. Also, on-chip power switches are highly effective for standby power reduction.

Figure 1 shows a die plot of Merlot, and Figure 2 depicts its block diagram. There are two integer-media pipelines (VL and VS) in each

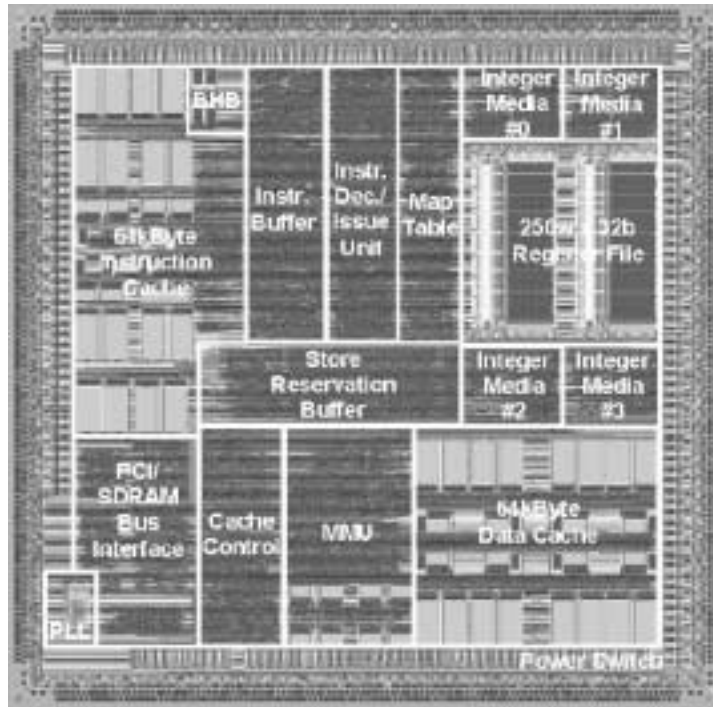


Figure 1. Merlot microprocessor, a single-chip MP98 prototype, offers 1-GIPS performance while consuming 1 watt of power.

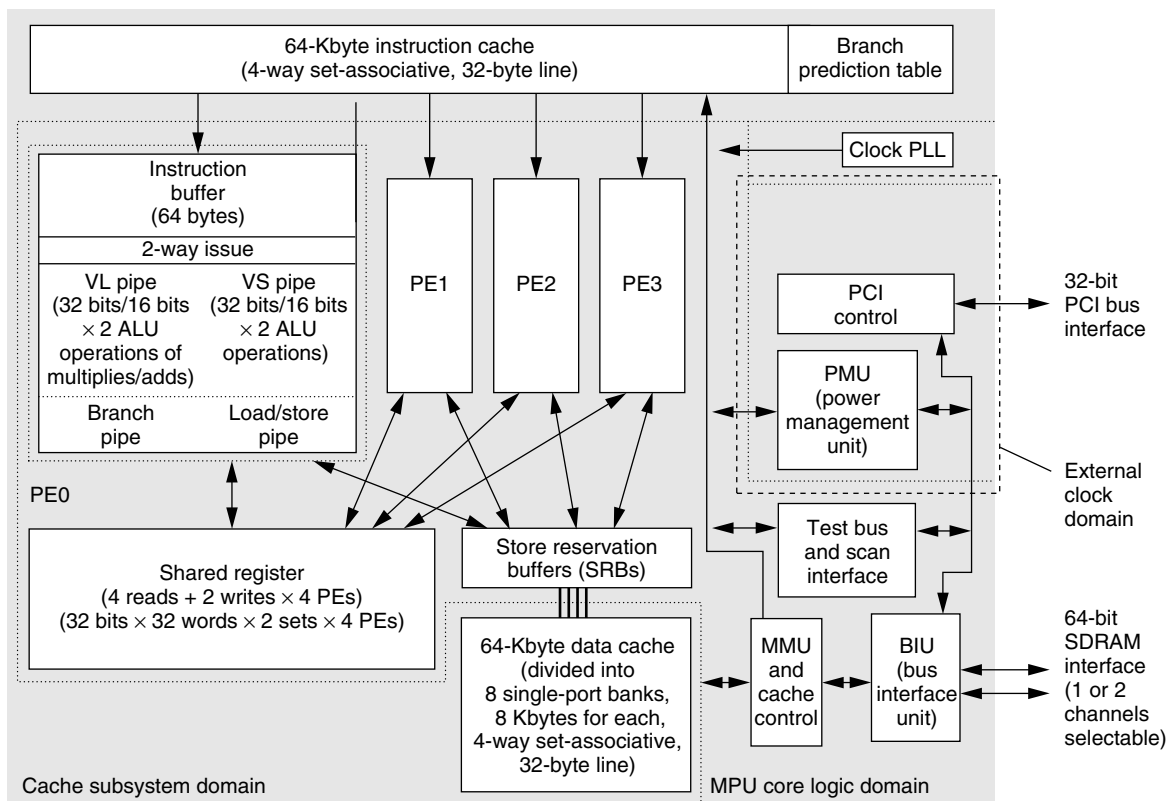


Figure 2. Processor block diagram.

processing element (PE). Both pipelines also have 32-bit or 16-bit  $\times$  2 SIMD ALUs. The VL pipe is equipped with a 32-bit divider, and 32-bit or 16-bit  $\times$  2 SIMD multiplier and multiply-and-add units. Also, each processing element has load/store and branch pipelines.

In each processing element, two instructions are fetched at a time from the 64-Kbyte, shared instruction cache through the instruction buffer then issued to the pipelines. Four processing elements share 16-read, 8-write register files, which achieve high-performance register-value inheritance at thread creation.

Data to be stored in memory is written in a 64-Kbyte, shared data cache through the store reservation buffer (SRB). In addition, Merlot has an external PCI interface and two channels of an external 64-bit SDRAM interface. This greater-than-1-Gbyte/sec SDRAM interface is extremely important in achieving high performance.<sup>3</sup>

The dotted lines in Figure 2 indicate the power control domains. On-chip power switches in Merlot control the microprocessor unit (MPU) core logic and cache subsystem domains separately. The switches can be turned off by operating systems and controlled by the power management unit (PMU) connecting with the PCI.

Estimates show that Merlot will perform 1 GIPS with 1-W power consumption during 1.3-V operations. When all power switches are turned off, the power consumption is estimated to be less than 100 microW. Also, note that users as well as the operating system can halt each processing element independently, so that only one processing element may operate for noncritical jobs. This power efficiency is extremely suitable for smart information terminals, which require a long battery life both in working and standby modes.

To obtain such excellent power efficiency, we need to extract parallelism from the software. MP98 uses a multithreaded software model suitable for fine- to coarse-grain thread parallelism. Using this software model, the MP98 hardware architecture implements features that support thread parallelism efficiently. With its on-chip, tightly coupled multiprocessor architecture, Merlot supports 1-cycle thread creation. This makes the fine-grain thread creation very practical. We can even use threads of about 10 instructions to achieve efficient parallel exe-

cutation. (Usually, such short threads are not worth creating because the thread creation overhead cancels performance gain.) Also, Merlot supports control and data-dependency speculation for performance acceleration.

### Multithreaded software model

In MP98, multiple control flows in a program code run concurrently on multiprocessors. This parallel execution model is called multiple control flow execution (MCFE). The MCFE in MP98 helps compilers extract parallelism from software with the following hardware support:

- Fork-once parallel execution:<sup>4</sup> FOPE is a special MCFE model proposed in the MP98. In the MCFE MP98 model, each thread creates at most a single thread called FOPE. With the FOPE model, threads are created one after another. The FOPE model reduces the amount of hardware and facilitates compiler analysis of control and data dependencies among threads as well.
- Control and data-dependency speculation: Even when compilers cannot guarantee parallel execution of threads without any dependency, MP98 can execute these threads in parallel using speculative execution.

Figure 3 shows our multithreaded software model. Figure 3a is an example of the control flow graph of a program, where A to E are basic blocks. This example shows a loop structure of a program in which the program control starts from A, iterates B to D, and ends at E.

Figure 3b depicts an execution flow on superscalar processors. Here, it is assumed that the loop is iterated three times, and basic blocks are named B1 to D3 in the flow graph. In superscalar processors several instructions are fetched at each clock, simultaneously executable instructions are selected at that time, and then these instructions are issued to execution units. This type of execution model is called instruction-level parallelism (ILP).

Figure 3c shows an MCFE execution model in MP98. In this example, basic blocks B to D are bundled in a thread. At the start of each iteration cycle (the starting basic block B in each thread), the thread creates another thread

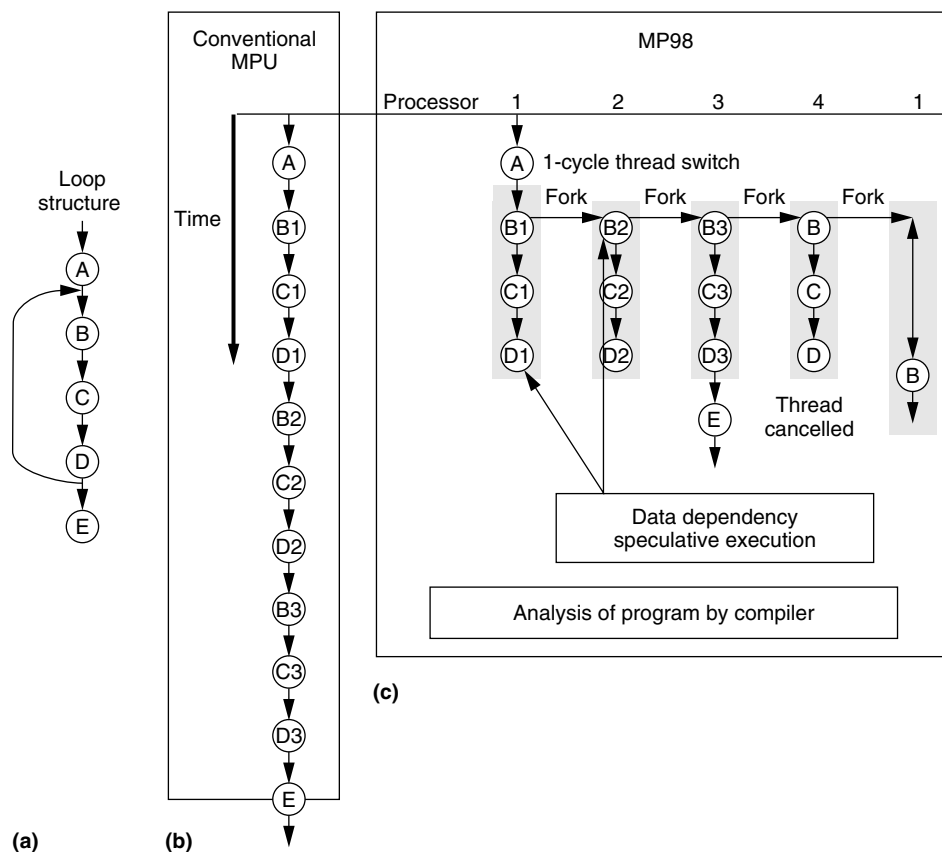


Figure 3. MP98 multithreaded software model: a program's control flow graph (a), execution flow on superscalar processors (b), and MCFE execution model in MP98 (c).

that is executed on the next processing element. This operation is called a fork. In our example, B2 is created just after the start of B1, B3 is created just after the start of B2, and so on. Since threads are executed in parallel, MCFE achieves more performance than ILP. Note that in MP98, software determines all fork positions in the program code.

A data dependency problem could be raised in this multithreaded model. For our example, with MCFE B2 may be executed before D1, which is the opposite order from the original program. If B2 uses a datum in memory calculated in D1, B2 needs to wait for the calculation in D1. To have B2 wait for a data access in memory, we can block the memory address in MP98. When the data value is set, D1 will release the address so that B2 uses it. This block-release mechanism is implemented in store reservation buffers and data cache systems.<sup>4</sup>

To bring out the maximum performance in applications, we defined directive codes,<sup>5,6</sup>

which are compiler directives in program source. They are designed to extract parallelism from program codes with single-chip multiprocessors like the MP98, such as forks with and without speculations and blocks of memory access. Also, we have a set of higher level directive codes for loop parallelization.

### Artificial intelligence and dictionaries

Since we expect that applications for artificial intelligence, especially dictionary handling and searching, will require the highest performance in future information terminals, we use methods to parallelize them for MP98. Here, we use the word *dictionary* in a general meaning. Methods could be a set of relations implemented with hash tables, a set of words on search trees, a set of points in a multidimensional space, and so on.

As we mentioned earlier, intelligence in information terminals requires recognition, understanding, imagination, and creation.

```

hash_main:
  # e and v are thread local variables.
  # A is a hash table.
  get next element e;
  fork.d hash_main; #fork next thread with
                        data speculation
  v := hash_function(e);
  put e in hash table at entry A[v];
  term; # terminate the thread (self)

```

Figure 4. Hashing algorithm with directive codes (underlined).

Basic recognition operations look up preconstructed dictionaries. For example, in speech recognition, after phonemes are extracted from the speech source, the phonemes are combined into syllables, word candidates are found from the syllables, and syntactic information is extracted. Dictionaries are used at each step in this procedure.

Semantic understanding information is extracted from a sequence of syntactic information using a knowledge base, a type of dictionary. Knowledge ought to be a huge amount of data, for example, relations among syntactic or background information. In addition, knowledge bases should be adaptive; that is, knowledge should be updated from time to time based on experiences. Handling, investigating, and updating these dictionaries require much higher CPU performance than recognition.

We observed that, in many cases, something imagined or created is not totally new, but it is just a new relation on existing dictionaries with an association. Even for these cases, since a lot of associated, nonrelated elements are tried, much more CPU performance is needed than for understanding in which new relations are given from experiences.

Merlot has an on-chip SDRAM interface with greater than a 1-Gbyte/sec throughput and less than 100 ns of cache miss penalty.<sup>3</sup> This performance is highly effective in handling large dictionaries.

### Algorithms, multithreaded software model

Although actual handling or searching algorithms depend on specific dictionaries, they should be implemented by a combination of basic algorithms such as hashing, sorting, and searching.<sup>7-10</sup>

### Hashing

Hashing, one of the most essential algorithms for dictionaries, is frequently used for symbol tables and is an efficient implementation method.

Clearly, tasks of looking up hash tables can be executed completely in parallel. However, it is not easy to parallelize tasks to put elements in hash tables because it is possible that two or more tasks simultaneously write a memory position, and this may cause incorrect results. Thus, hashing cannot be parallelized in many parallel-processing models.

The good news for MP98 is that the possibility of such hazards is very low. For a hash table of size  $N$ , the probability of conflict is  $1/N$  for uniformly distributed data. Therefore, when all tasks are executed in parallel in a speculation mode, the probability of the failure of the speculation is only  $1/N$ . Since the recovery cost is inexpensive in MP98, much parallelism can be extracted from hashing.

Figure 4 provides a pseudocode for a basic hashing algorithm with our directive codes.<sup>5,7,9,10</sup> To explain how the code works, we assume that thread  $t_1$  undertakes a task to put element  $e_1$  in hash table A. As soon as data element  $e_1$  is read,  $t_1$  creates new thread  $t_2$  for next element  $e_2$ . Data speculation is used for the fork, so that  $t_2$  continues its execution, but all data stored by  $t_2$  are kept in store reservation buffers and wait for termination of  $t_1$ . If it is detected (in data cache systems) that  $t_2$  has loaded incorrect data from memory—that is,  $t_1$  writes data in memory after  $t_2$  has read— $t_2$  restarts. Thus, both threads are concurrently executed, and  $t_2$  can write data in memory only after  $t_1$  terminates. As mentioned before, data speculation rarely fails in hashing; and therefore, hashing is efficiently parallelized in multithreaded architectures like MP98.

### Sorting

Sorting is also a fundamental algorithm in actual applications.<sup>7,9,10</sup> Parallel sorting has been analyzed thoroughly for more than 30 years, and the odd-even merge sort is one of the oldest but most famous algorithms.<sup>11</sup> However, when the number of processing elements is a small constant such as four (in Merlot), this algorithm is impractical because the time complexity will be  $O(n \log^2 n)$  for  $n$  numbers. Among  $O(n \log n)$  sorting algorithms,

## Sorting

Our algorithm has three steps: 1) to divide the number list to be sorted into several (typically, the number of processing elements) sublists, 2) to use a quick sort for each divided list, and 3) to complete sorting using a merge sort. Figure A depicts our sorting algorithm with a small example. Figure A1 shows a given number list to be sorted, and we assume that the number of processing elements is four. This list is subdivided into four sublists for four processing elements, and each sublist is assigned to a processing element and sorted by a quick sort (Figure A2).

Before we complete the sorting, we present a method to merge two sorted sublists  $S$  and  $T$  with  $q$  processing elements. First, determine  $(q-1)$  intermediate positions (of an equal-size interval usually) on  $S$  (Figure A3). Then, find associated  $q-1$  positions on  $T$  with  $q-1$  processing elements, where the associated position with position  $s$  on  $S$  is position  $t$  such that  $T[t] \leq S[s] < T[t+1]$  (if there is such a  $t$  on  $T$ ) (Figure A4). Lastly, sublists on  $S$  and  $T$  subdivided by the intermediate or associated positions are merged with  $q$  processing elements (Figure A5).

Now, we complete the sorting with Figure A1. First, two sets of two

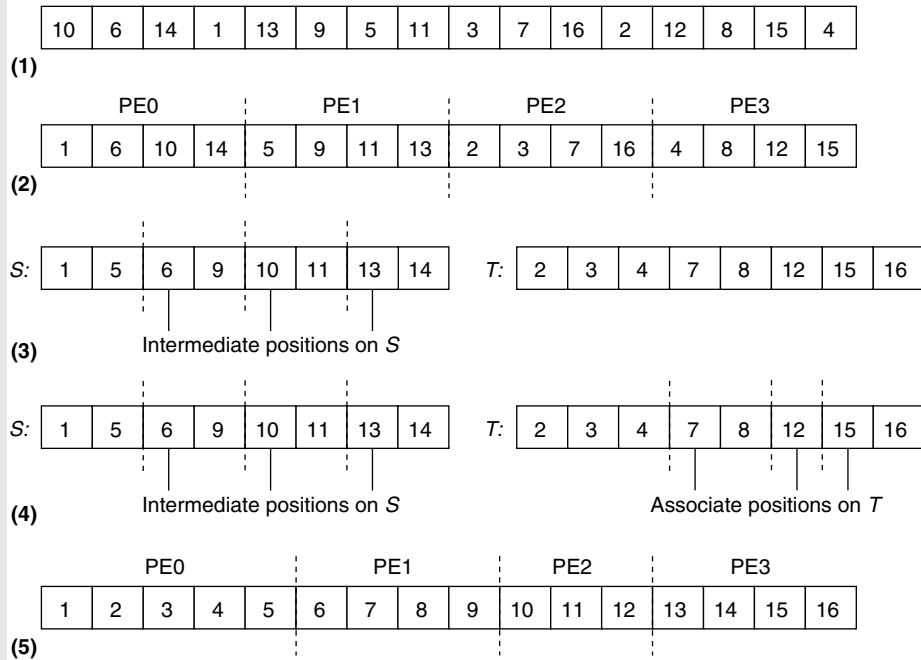


Figure A. Sorting algorithm example: given numbers (1), sorting four sublists with four PEs (2), two sorted sublists and intermediate positions on  $S$  ( $q = 4$ ) (3), associate positions on  $T$  (4), and the sorted list (5).

sorted sublists are merged with four processing elements. Note that two sorted lists after the merge are equivalent to the sorted lists as shown in Figure A3. Finally, we obtain the final sorted list as we just explained (Figure A5). Clearly, the time complexity is  $O(n \log n)$ .

the quick sort is not easy to parallelize until several subproblems for sorting are generated. Although the heap sort can be parallelized with a pipelining technology, heap sort is essentially slower than quick sort.

In our case, an efficient algorithm has three steps: 1) to divide the number list to be sorted into several (typically, the number of processing elements) sublists, 2) to use a quick sort for each divided list, and 3) to complete sorting using a merge sort. Clearly, the time complexity is  $O(n \log n)$ ; all three steps are easily parallelized. Details are shown in the “Sorting” box.

### Tree search

A tree is another common data structure for dictionaries. There are two typical search methods for trees: depth- and breadth-first

searches.<sup>7,8,10</sup> Although both algorithms can be parallelized in MP98, we describe only the breadth-first search because our speech recognition algorithm uses it.

The basic operations of a breadth-first search method are 1) to take the first node on a queue and 2) to add nodes adjacent to the node to the queue. When positions on the queue used by a thread are reserved in a timely fashion by the thread, the breadth-first search is easily parallelized. An example is shown in the “Breadth-first search” box (next page).

### Multidimensional search

Multidimensional data structures are frequently used for applications in which the distance between data on a multidimensional space represents their relation. For example,

### Breadth-first search

Basic operations in breadth-first searches are 1) put the root node in a queue; 2) if the queue is empty, end the program, otherwise, take a node from the queue; 3) do some jobs for the node; and 4) add child nodes to the queue and go to step 2. Here, we assume that there is no internode dependency in step 3 and that each node  $n_i$  is associated with number of children  $c_i$ .

Parallelism can be extracted from the loop of steps 2 through 4. Figure B shows our parallelized pseudocode. In our code, global array variable  $Q$  denotes the queue. In the search\_main thread, first, take a node from the queue. Note that we use data speculation mode for this thread again because the node may be read before its parent writes the node in the queue. Then, reserve positions on  $Q$  for children of the node, and fork the next thread. After the fork operation, add its child nodes to the queue, and its job is processed.

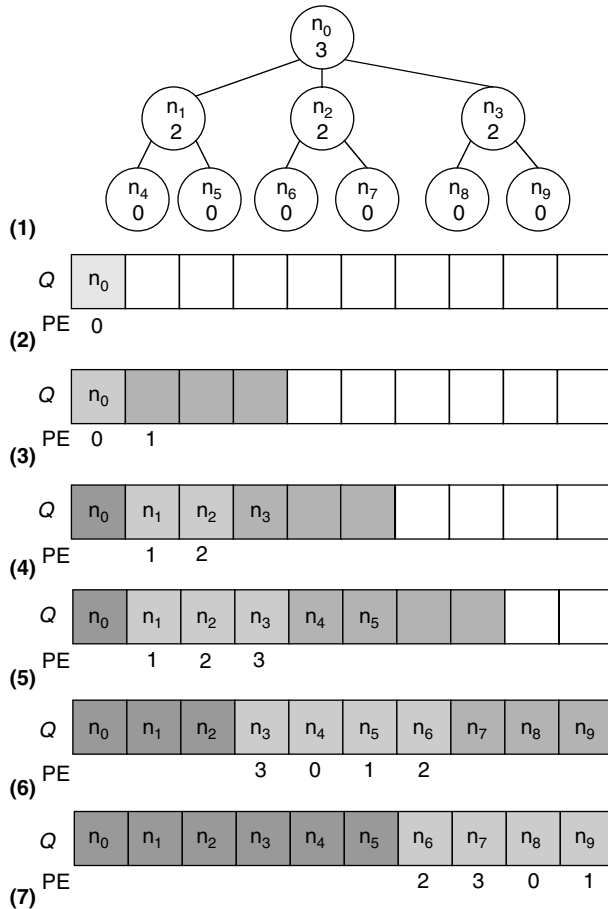
Figure C shows an example of a parallelized breadth-first search with a small tree of 10 nodes  $n_0$  to  $n_9$  (Figure C1). In the tree, numbers under the node names are  $c_i$ , the number of children of the node. First, node  $n_0$  is processed at PE0, processor element 0 (Figure C2). After taking node  $n_0$  and reserving three positions on  $Q$ , the next thread starts on PE1 (Figure C3). However, since the next position on  $Q$  is reserved but is not added to any node yet, the thread waits until a node is added. After the thread for  $n_0$  adds nodes  $n_1, n_2,$  and  $n_3$  to the queue, the thread for  $n_1$  starts again (Figure C4). This thread creates a new thread for  $n_2$  and adds  $n_4$  and  $n_5$  to the queue (Figure C5). Then, threads successively start, as shown in Figures C6 and C7.

Figure C. Example breadth-first search assuming four processing elements: sample tree (1), starting the first thread (2), after the first fork (3), after the second fork (4), after the third fork (5), after the sixth fork (6), and after the final fork (7).

```

start:
  # Q is global array variable for Queue.
  # n is thread local variable.
  add the root  $n_0$  to Q;
search_main:
  if (Q is empty) exit;
  get next node n from Q;
  reserve positions of Q for children;
  fork.d search_main; # fork next thread
  for (each child v of node n) {
    add v to a reserved position of Q;
  }
  do job for node n;
  term; # terminate the thread
    
```

Figure B. Basic breadth-first search algorithm with directive codes (underlined).



a dictionary for people with their height, weight, and age data can be represented as a 3D search structure. A query could be, "Find

a group of suspicious persons about 6 feet, 160 pounds, and 30 years old."

Bucketing is a technique for multidimen-

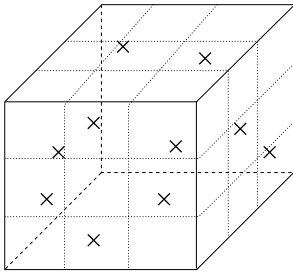


Figure 5. Bucket structure for multidimensional searches.

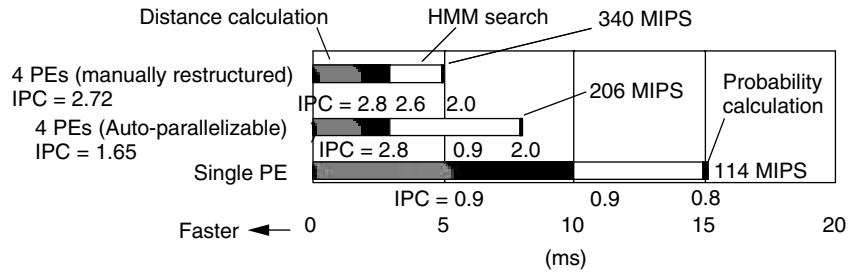


Figure 6. Estimated results for a speech recognition program.

sional data structures that is easy to use. We assume that the distance space used in the applications concerned is covered by a multidimensional rectangle, and bucketing is a method to subdivide the rectangle into equal-size multidimensional rectangles (buckets) by meshes (Figure 5). For a query point in the distance space, it is very easy to determine which bucket contains the point. Bucket searching is efficient when searching for the most related (nearest) point because we need to search only for buckets around the point.

In the bucketing technique, the tasks to put data into bucket structures can also be executed in parallel with MP98 just as in the hash techniques. Also, the tasks to find related points for given points can be performed concurrently.

### Estimating speech recognition

Here, we report our estimation results for parallelization of a speech recognition program code that is internally implemented at NEC. The program is written in the C language with compiler directive codes and compiled with our C compiler. The execution time is estimated with our pipeline-level simulator, and our power estimation tool estimates the power dissipation. Input data of the speech recognition program are speech feature parameters extracted from an actual speech example, and outputs are words. Most of the execution time is spent in three functions: Gaussian probability calculation (probability calculation), Hidden Markov Model (HMM) state probability calculation (distance calculation), and HMM Viterbi search.<sup>12</sup>

The probability and distance calculation finds similarities between inputs and phonemes in the dictionary. Our program code consists of simple loop structures with some branch and

max/min operations. Our compiler parallelizes this program, though currently some directive codes are added.

The HMM search finds word candidates based on state probabilities. To search for candidates, HMM traverses a large dictionary tree for about 100,000 words. We use a variation of the beam search algorithm<sup>13</sup> in the dictionary search, which is similar to the breadth-first search but hopeless nodes are cut off at each step. With the parallel breadth-first search algorithm, our HMM search is also parallelized.

As shown in Figure 6, we estimated our parallelized code (currently with the aid of directive operations) to achieve about twice the speed of code for a single processor. Also, when a part of the code is rewritten with parallel algorithms, we increase speed three times. We estimate that the power dissipation is 1.1 W, even for this high-performance case.

To obtain high performance, we can use two methods: high frequency and high parallelism. It seems that frequency increases will be slower in the 21st century. Although a parallel architecture requires less power than a high-frequency architecture, its highest power efficiency is achieved when high parallelism is extracted from applications. Since algorithms used in artificial intelligence are effectively parallelized on multithreaded software models, single-chip, high-performance, low-voltage multiprocessors like the MP98 are promising for future information terminals that demand mobility, intelligence, and diversity.

It is plausible that microprocessors at the end of the 21st century will have different styles such as biocomputers. Even for such new computation models we believe parallelism, including multithreaded models for integrat-

ed parallel processors, is an indispensable key issue for future microprocessors.

MICRO

### Acknowledgments

We are grateful to Tatsuo Ishiguro, Hirokazu Goto, Hisatsune Watanabe, Satoshi Goto, Masao Fukuma, and Takao Nishitani for their constant encouragement. Also, we thank Akihiko Konagaya of JAIST and all members of the MP98 project for their valuable discussions. Ryosuke Isotani greatly helped us in implementation and analysis of speech recognition applications.

### References

1. L. Geppert and W. Sweet, "Technology 2000: Analysis & Forecast," *IEEE Spectrum*, Jan. 2000, pp. 27-31.
2. E. Sumita, N. Nisiyama, and H. Iida, "The Relationship between Architectures and Example-Retrieval Times," *Proc. 12th Nat'l Conf. Artificial Intelligence, AAAI'94*, American Assoc. for Artificial Intelligence.
3. S. Matsushita et al., "Merlot: A Single-Chip Tightly Coupled Four-Way Multi-Thread Processor," *Proc. Cool Chips III Symp.*, Apr. 2000, Kikai-Shinko-Kaikan, Tokyo, pp. 63-74.
4. N. Nishi et al., "A 1GIPS 1W Single-Chip Tightly-Coupled Four-Way Multiprocessor With Architecture Support for Multiple Control Flow Execution," *Proc. Int'l Solid-State Circuits Conf. (ISSCC 2000)*, IEEE Press, Piscataway, N.J., Feb. 2000, pp. 418-419.
5. J. Sakai et al., "Software Environment for Single-Chip Multi-Processor Merlot (MP98 Ver. 1)," *Proc. Cool Chips III Symp.*, Kikai-Shinko-Kaikan, Apr. 2000, p. 277.
6. MP98 Web site: <http://www.labs.nec.co.jp/MP98/>, 2000.
7. A.V. Aho, J.E. Hopcroft, and J.D. Ulman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Mass., 1976.
8. D.E. Knuth, *The Art of Computer Programming: Fundamental Algorithms*, Addison-Wesley, 1968.
9. D.E. Knuth, *The Art of Computer Programming: Sorting and Searching*, Addison-Wesley, 1973.
10. R. Sedgewick, *Algorithms in C*, Addison-Wesley, 1990.
11. F.T. Leighton, *Introduction to Parallel Algorithms and Architectures: Arrays-Trees-Hypercubes*, Morgan Kaufmann, San Mateo, Calif., 1992.
12. L.R. Rabiner and B.H. Juang, "An Introduction to Hidden Markov Models," *IEEE ASSP Magazine*, Jan. 1986, pp. 4-16.
13. P.H. Winston, *Artificial Intelligence*, 3rd ed., Addison-Wesley, 1992.

**Masato Eda**hiro is a principal researcher at System Devices and Fundamental Research, NEC Corporation, where he participates in MP98 software development environments and application analysis. Edahiro received BS and MS degrees in mathematical engineering from the University of Tokyo and MA and PhD degrees in computer science from Princeton University. He is a member of the IEICE, IEICI, ORSJ, and IPSJ.

**Satoshi Matsushita** is a principal researcher at System Devices and Fundamental Research, NEC Corporation, where he is in charge of CPU RTL design and logic verification for MP98. He also helped in the development of the MIPS RISC processors. Matsushita received BS and MS degrees in electrical engineering from the University of Tokyo. He is a member of the IEEE and the IPSJ.

**Masakazu Yamashina** is a senior manager at System Devices and Fundamental Research, NEC Corporation. He manages research on high-performance and low-power architecture and circuits. Yamashina holds BS, MS, and Dr Eng degrees from the Tokyo Institute of Technology. He is a senior member of the IEEE, a member of the IEICE, and the program chair for the VLSI Circuit Symposium.

**Naoki Nishi** is a project manager at System Devices and Fundamental Research, NEC Corporation, where he leads the MP98 Project team. Nishi received BS and MS degrees in system engineering from Hiroshima University. He is a member of the IEICE and the IPSJ.

Direct comments to Masato Edahiro, System Devices and Fundamental Research, 24-23070, NEC Corporation, 1120 Shimokuzawa, Sagami-hara, Kanagawa 229-1198, Japan; eda@bp.jp.nec.com.